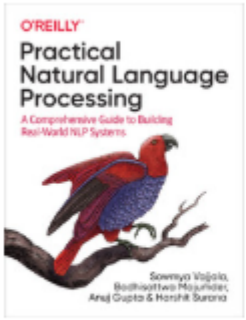


Practical Natural Language Processing

Sowmya Vajjala, Bodhisattwa Majumder, Anuj Gupta, Harshit Surana

Published by O'Reilly Media, Inc.

# Chapter 11. The End-to-End NLP Process

*The process is more important than the goal. The person you become is infinitely more valuable than whatever the result is.*

*Anthony Moore*

So far in the book, we've addressed a range of NLP problems, starting from what an NLP pipeline looks like to how NLP is applied in different domains. Efficiently applying what we've learned to build end-to-end software products involving NLP takes more than just stitching together various steps in an NLP pipeline—there are several decision points during the process. While a lot of this knowledge comes only with experience, we've distilled some of our knowledge about the end-to-end NLP process in this chapter to help you hit the ground running faster and better.

In Chapter 2, we already saw what a typical pipeline for an NLP system looks like. How is this chapter then any different from that? In Chapter 2, we focused primarily on the technical aspects of the pipeline—for example, how do we represent text? What pre-processing steps should we do? How do we build a model, and then how do we evaluate it? In the subsequent chapters in Parts I and II of the book, we delved deeper into different algorithms to perform various NLP tasks. We also saw how NLP is used in various industry domains, such as healthcare, e-commerce, and social media. However, in all these chapters, we spent little time on the issues related to deploying and maintaining such systems and on the processes to follow when managing such projects. These are the focus of this chapter. Most of the points discussed here are broadly applicable not just to NLP, but also to other concepts, such as data science (DS), machine learning, artificial intelligence (AI), etc. Throughout this chapter, we use these terms interchangeably; where the focus is specifically on NLP tasks, we mention that explicitly.

We'll start the discussion by revisiting the NLP pipeline we introduced in Chapter 2 and take a look at the last two steps: deployment, followed by monitoring and updating the model, which we didn't cover in earlier chapters. We'll also see what it takes to build and maintain a mature NLP system. This is followed by a discussion on the data science processes followed in various AI teams, especially with respect to building NLP software in particular. We'll conclude the chapter with a lot of recommendations, best practices, and

do's and don'ts to successfully deliver NLP projects. Let's start by looking at how to deploy NLP software.

Revisiting the NLP Pipeline: Deploying NLP Software

In Chapter 2, we saw that a typical production pipeline for NLP projects consists of the following stages: data acquisition, text cleaning, text pre-processing, text representation and feature engineering, modeling, evaluation, deployment, monitoring, and model updating. When we encounter a new problem scenario involving NLP in our organization, we have to first start thinking about creating an NLP pipeline covering these stages. Some of the questions we should ask ourselves in this process are:

- What kind of data do we need for training the NLP system? Where do we get this data from? These questions are important at the start and also later as the model matures.
- How much data is available? If it's not enough, what data augmentation techniques can we try?
- How will we label the data, if necessary?
- How will we quantify the performance of our model? What metrics will we use to do that?
- How will we deploy the system? Using API calls over the cloud, or a monolith system, or an embedded module on an edge device?
- How will the predictions be served: streaming or batch process?
- Would we need to update the model? If yes, what will the update frequency be: daily, weekly, monthly?
- Do we need a monitoring and alerting mechanism for model performance? If yes, what kind of mechanism do we need and how will we put it in place?

Once we've thought through these key decision points, a broad design of our pipeline is ready! We can then start to focus on building version 1 of the model with strong baselines, implementing the pipeline, deploying the model, and from there, iteratively improving our solution. In Chapter 2, we saw how different stages of the NLP pipeline before deployment are implemented for various NLP tasks. Let's now take a look at the final stages of the pipeline: deployment, monitoring, and model updating.

What does deployment mean? Any NLP model we build is typically a part of some larger software system. Once our model is working well in isolation, we plug it into a larger system and ensure that everything is working well. The set of all of the tasks related to integrating the model with the rest of the software and making it production-ready is called *deployment*. Typical steps in deployment of a model include:

1. *Model packaging:* If the model is large, it might need to be saved in persistent cloud storage, such as AWS S3, Azure Blob Storage, or Google Cloud Storage, for easy access. It might also be serialized and wrapped up in a library call for easy access. There are also open formats like ONNX [1] that provide interoperability across different frameworks.

2. *Model serving:* The model can be made available as a web service for other services to consume. In cases where a more tightly coupled system and batch process is more applicable, the model could be part of a task flow system like Airflow [2], Oozie [3], or Chef [4], instead of a web service. Microsoft has also released reference pipelines for MLOps [5] and MLOps in Python [6].

3. *Model scaling:* Models that are hosted as web services should be able to scale with respect to request traffic. Models that are running as part of a batch service should also be able to scale with respect to the input batch size. Public cloud platforms as well as on-premise cloud systems have technologies that enable that. Figure 11-1 shows one such pipeline for text classification on AWS. More details on the engineering of this pipeline can be found in the AWS post [7].

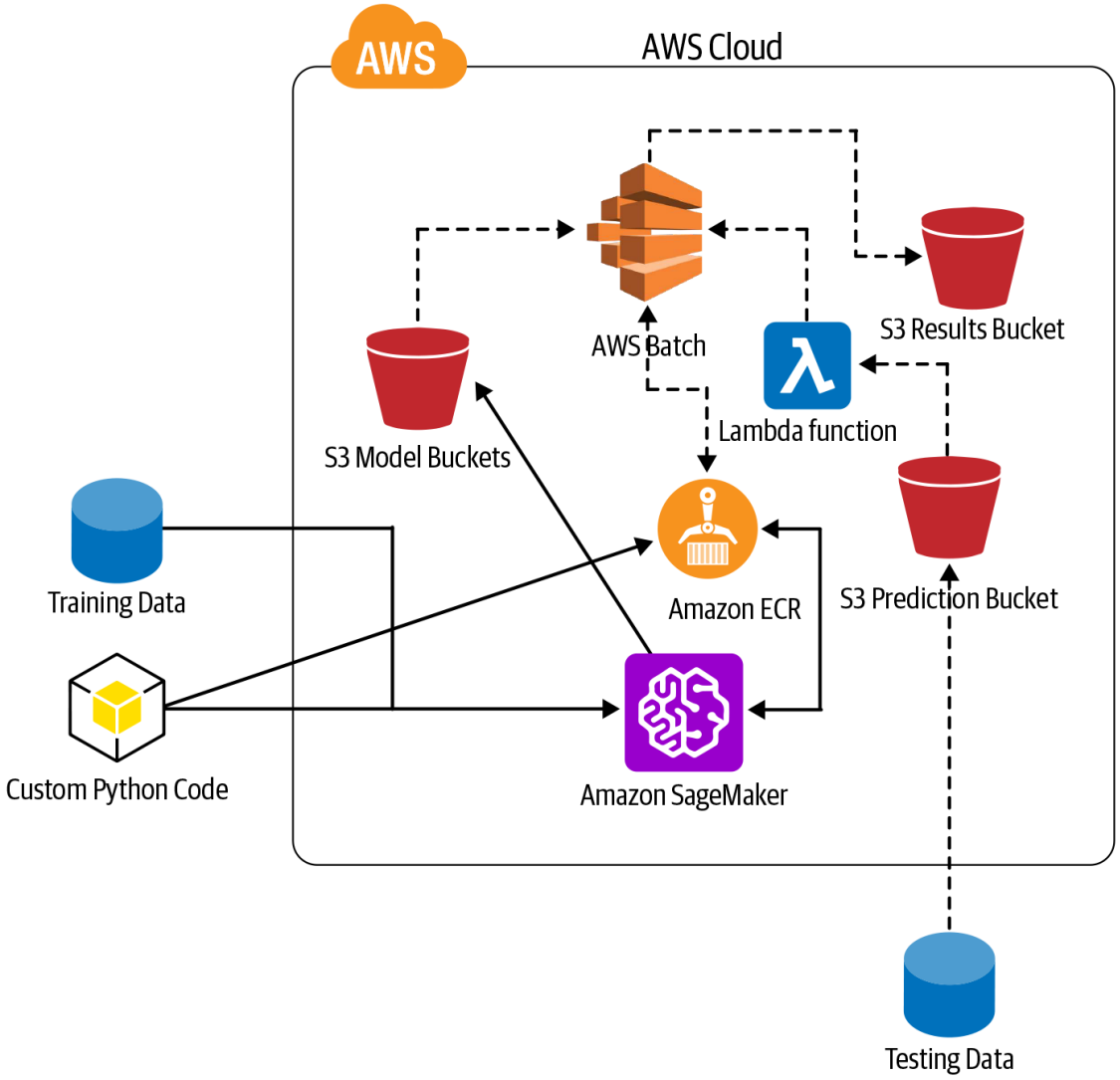


Figure 11-1. AWS Cloud and SageMaker to serve text classification [8]

Let's look at an example to understand the deployment of an NLP model into a larger system.

**An Example Scenario**

Let's say we work for a social media platform and are asked to build a classifier to identify abusive user comments. The goal of this classifier is to prevent abusive content from

appearing on the platform by flagging any content that's potentially abusive and sending it for human moderation. We worked hard on collecting the data relevant to this task, designing a set of features, and testing a range of algorithms, and we built a predictive model that takes a new comment as input and classifies it as abusive or safe. What next?

Our model is just a small part of the larger social media platform. There are several components: content is being rendered dynamically, and there are various modules to interact with users, components responsible for storage and retrieval of data, and so on. It's possible that different subsystems of the platform are written in different programming languages. Our classifier is just a small component of the product, and we need to integrate it into the larger setup. How do we go about doing this? A common way to address this scenario is to create a web service where the model sits behind the web service. The rest of the product interacts with the model via this web service. It queries the service with the new comment(s) and gets back the prediction(s). The call to this web service is integrated into the product wherever necessary. Popular web application frameworks such as Flask [9], Falcon [10], and Django [11] are typically used to create such web services.

Developing various NLP solutions involves relying on a range of pre-existing libraries. Setting up a web service and hosting what we built in the cloud or some server requires us to ensure that there are no compatibility issues. To address this, there is a range of options available. The most common option is to package various libraries into a container like Docker [12] or Kubernetes [13]. Operationalizing a web service for production requires addressing many other issues, such as tech stack, load balancing, latency, throughput, availability, and reliability. Building and making a model production ready includes a whole lot of engineering tasks, which can often be time consuming. Cloud services such as AWS SageMaker [14] and Azure Cognitive Services [15] try to make these engineering tasks easy. Sometimes, the whole process, to the last detail, is automated to such an extent that it's as simple as one-click-get-done to set up the service. The idea is to let the AI teams focus on the most important part: model building.

Another important issue to address is model size. Modern NLP models can be quite large. For example, Google's Word2vec model is 4.8 GB in size and takes over 100 seconds just to load into memory (refer back to *Ch3/Pre_Trained_Word_Embeddings.ipynb*). Likewise, a fastText classification model is typically over 2 GB in size. DL models like BERT are known to be even bulkier. Hosting such large models in the cloud can be both challenging and expensive. There's a lot of work happening in the area of model compression to address such scenarios. Some of them are listed below:

- "Compressing BERT for Faster Prediction," a blog post by a team at Rasa NLP [16]
- "A Survey of Model Compression and Acceleration for Deep Neural Networks," a report by a team at Microsoft Research and Tsinghua University [17]
- "FastText.zip: Compressing text classification models," a report by a team at Facebook AI Research [18]
- "Awesome ML Model Compression," a GitHub repository by Cedric Chee that includes relevant papers, videos, libraries, and tools [19]

This is just a brief overview of various steps that go into deploying our NLP model. There are books and other materials that cover this in complete detail. As a start, interested readers can look at the later chapters of the book *Machine Learning Engineering* [20].

For most industry use cases, model building is seldom a one-time activity. As the deployed system gets used more, the models built need to adapt to new scenarios and new data points. Hence, the models should be updated regularly. Let's discuss the issues to consider while building and maintaining mature NLP software.

Building and Maintaining a Mature System

In most real-world settings, the underlying patterns in data change over a period of time. This means that the models that were trained long before can become stale—i.e., the data used to train the model is very different from the data in the production environment that's being fed to the model for predictions. This is called *covariate shift*, and it results in a performance drop of the model. Model update is a common approach to deal with such scenarios. On a similar note, in most industrial settings, once the first version of a model is consumed, improving the model becomes inevitable. Updating and improving an existing NLP model could just mean retraining with newer or additional training data, or it sometimes involves adding new features. When updating such models, the goal is to ensure that the deployed system performs at least as well as the existing system. Most model updates and improvements lead to more complex models. As the models grow in complexity, we need to ensure that the system doesn't crumble under increasing complexity. We need to manage the complexity of a mature NLP model while making sure it's also maintainable. Some of the issues we need to consider in this process are:

- Finding better features
- Iterating existing models
- Code and model reproducibility
- Troubleshooting and testing
- Minimizing technical debt
- Automating the ML process

In this section, let's take a look at these issues one by one, starting with a discussion about how to find better features.

**Finding Better Features**

Throughout this book, we've repeatedly stressed the importance of building a simple model first. This version 1 model is seldom an end in itself. We may keep on adding new features and periodically retraining the model beyond V1. Our goal is to find the features that are most expressive to capture the regularities in the data that are useful for making predictions. How can we develop such features? We saw different ways to generate textual feature representations in Chapter 3. We can start with one of those that doesn't require prior knowledge about the problem domain (e.g., basic vectorization, distributed representations, and universal representations) or use our prior knowledge about the problem and domain to develop specific features for our problem (i.e., handcrafted features) or use a combination of both.

Designing specific features for a given problem (or feature engineering) can be both difficult and expensive. This is why problem-agnostic text representations are commonly used as a starting point. However, domain-specific features have their own value. For example, in a task of sentiment classification, more than vector representations of raw text, domain-specific indicators, such as count of negative words, count of positive words, and other word- and phrase-level features, are useful to extract the sentiment in a more robust manner.

Let's say we implemented a bunch of features to build our NLP models. Does the best model need each one of these features? How do we choose the most informative features among the several we implemented? For example, if we use two features where one can be derived from the other, we're not adding any extra information to the model. Feature selection is a great technique to handle such cases and make informed decisions. There are plenty of statistical methods that can be used to fine-tune our feature sets by removing redundant or irrelevant features. This broad area is called *feature selection*.

Two popular techniques for feature selection are wrapper methods and filter methods. Wrapper methods use an ML model to score feature subsets. Each new subset is used to train a model, which is tested on a hold-out set and then used to identify the best features based on the error rate of the model. Wrapper methods are computationally expensive, but they often provide the best set of features. Filter methods use some sort of proxy measure instead of the error rate to rank and score features (e.g., correlation among the features and correlation with the output predictions). Such measures are fast to compute while still capturing the usefulness of the feature set. Filter methods are usually less computationally expensive than wrappers, but they produce a feature set that's not as well optimized to a specific type of predictive model. In DL-based approaches, while feature engineering and feature selection is automated, we have to experiment with various model architectures.

Since feature selection methods are usually task specific (i.e., methods for classification tasks are different from methods for, say, machine translation), interested readers can look into resources such as sparse features, dense features, and feature interactions from Wide and Deep Learning from Google AI [21]. The book *Feature Engineering for Machine Learning* [22] would also be useful. However, we hope this overview convinced you of feature selection's role in building mature, production-quality NLP systems. Assuming we're going through this process of adding new features and evaluating them, how should we incorporate them into our training process and update our NLP models? Let's take a look at this question now.

**Iterating Existing Models**

As we mentioned earlier, any NLP model is seldom a static entity. We're often required to update our models even in production systems. There are several reasons for this. We may get more (and newer) data that differs from previous training data. If we don't update our model to reflect this change, it will soon become stale and churn out poor predictions. We may get some user feedback on where the model predictions are going wrong. This will then require us to reflect on the model and its features and make amendments accordingly. In both cases, we need to set up a process to periodically retrain and update the existing model and deploy the new model in production.

When we develop a new model, intuitively, it's always good to compare the results with our previous best models to understand the incremental value addition. How do we know this new model is better than the existing one? The analysis of model performance can be based on comparing raw predictions from both models, or it could be from a perspective of a derived performance based on the predictions. Let's explain these two cases by revisiting the abusive comments detection example from earlier in this chapter.

Let's say we have a gold standard test set of abusive versus non-abusive comments. We can always use this to compare an old model with the new one in terms of, say, classification accuracy. We can also follow an external validation approach and look for other aspects, such as how many model decisions were contested by users every day. It would be practical to set up a dashboard to monitor these metrics periodically and display them for each model so that we can choose the one that's the best improvement over the current model among the various models we may build. We can also A/B test a new model with an old model (or any baseline system) and measure business KPIs to see how well the new model performs. When onboarding a new model, it might also be a good practice to first roll it out to a small fraction of users, monitor its performance, and then progressively expand it to the entire user base.

**Code and Model Reproducibility**

Making sure your NLP models continue working in the same fashion in different environments can be critical for the long-term success of any project. A model or result that's reproducible is generally considered more robust. There is a range of best practices you can use to achieve this while building systems.

Maintaining separation between code, data, and model(s) is always a good strategy. Separating code and data is generally a best practice in software engineering, and it becomes even more critical for AI systems. While there are established version control systems for code, such as Git, versioning of models and datasets can be different. As of recently, there are tools like Data Version Control [23] that address this issue. It's always a good practice to name model and data versions appropriately so that we can revert back easily, if needed. While storing the models, you should try to have all your model parameters, along with other variables, in a separate file. Similarly, try to avoid hardcoded parameter values in your model. If you have to use arbitrary numbers in your training process (e.g., a seed value somewhere), explain it in the code as comments.

Another good practice is creating checkpoints in your code and model often. You should store your learned model in a repository both periodically and at milestones. When training a model, it's also a good idea to use the same seed wherever random initialization is used. This ensures that the model creates similar results (and internal representation) every time the same parameters and data are used.

A keystone for improving reproducibility is to make sure to note all steps explicitly. This is especially necessary in the exploratory phase of data analysis. On the same note, it helps to record as many intermediate steps and data outputs as possible. This helps in transforming your experimental model to an in-production model without any loss of information. To read further, we would suggest a report on AI reproducibility state of the art [24] and an interview of a reproducibility researcher at Facebook, Joelle Pineau [25]. This brings us to the next topic in this section. While making all these iterations and building

multiple models, how do we ensure there are no errors and bugs in the training process and that our data isn't noisy? How do we troubleshoot and test our code and models?

**Troubleshooting and Interpretability**

To maintain the quality of software, testing is a key step in any software development process. However, considering the probabilistic nature of ML models, how to test ML models is not obvious. Figures 11-2 and 11-3 illustrate some of the good practices for testing out AI systems. We already saw how to use Lime (Figure 11-3) in Chapter 4.

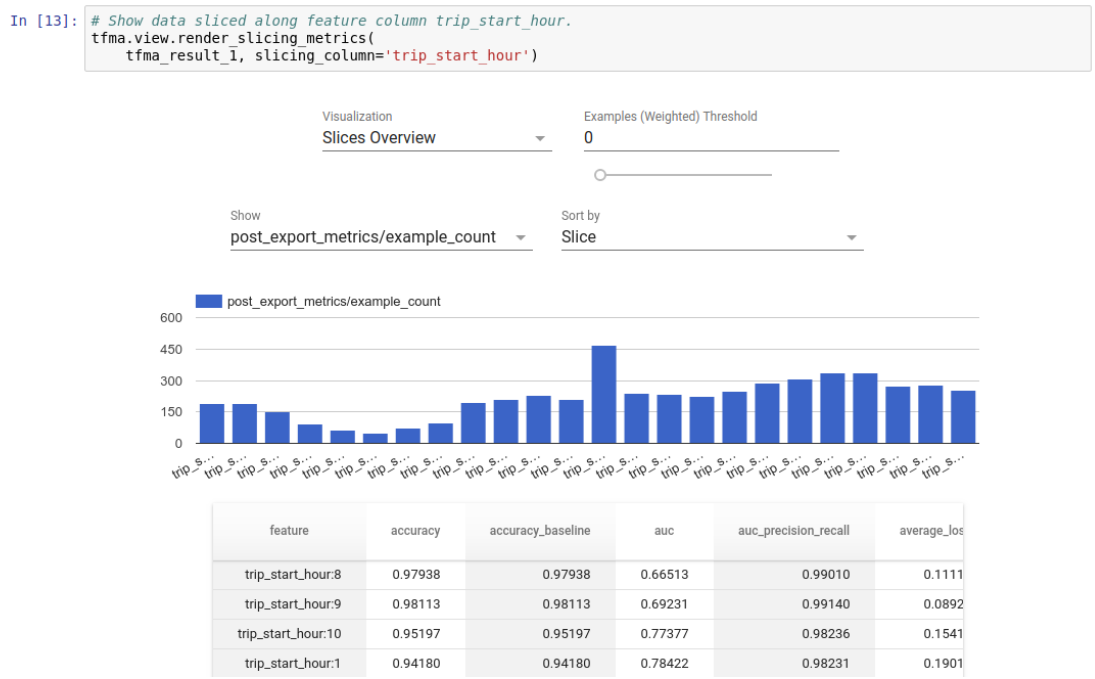


Figure 11-2. TensorFlow Model Analysis (TFMA) [26]

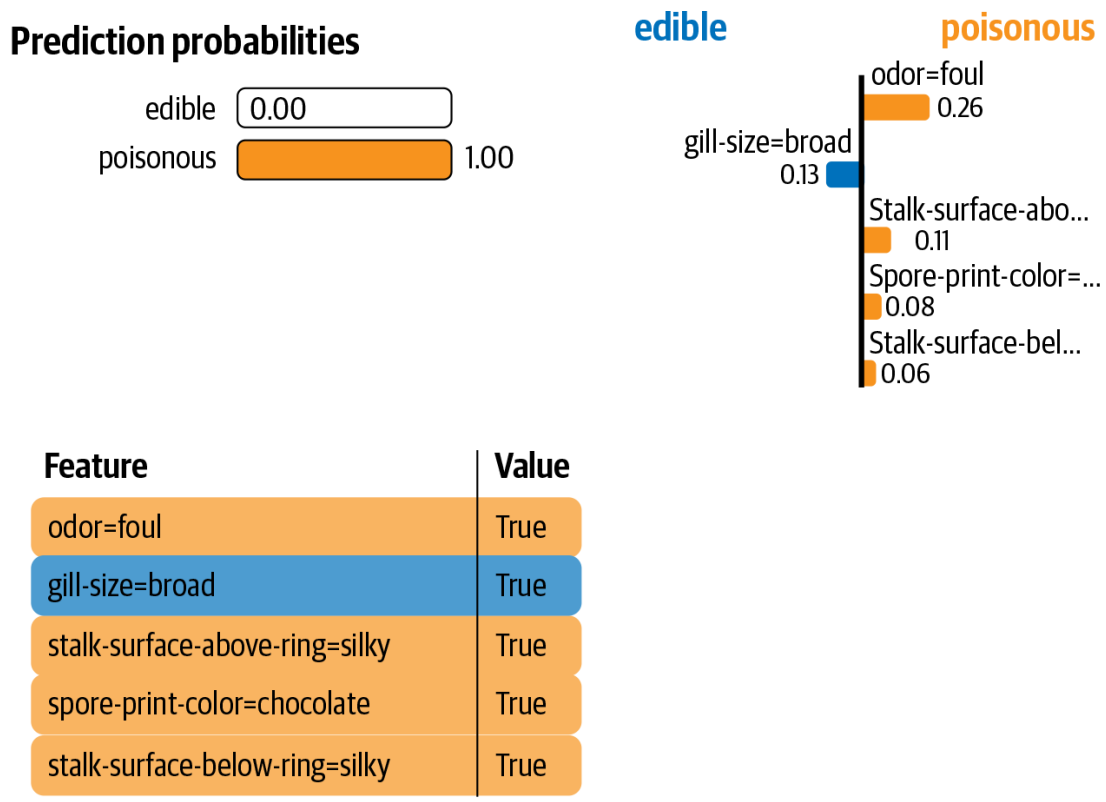


Figure 11-3. Lime for NLP model analysis

As we discussed earlier in the chapter, a model is just a small component of any AI system. When it comes to testing the entire system, barring the model, most techniques for testing of software engineering are applicable and work well. When it comes to testing the model, the following steps are helpful:

- Run the model on train, validation, and test datasets used during the model-building phase. There should not be any major deviation in the results for any of the metrics. K-fold cross validation is often used to verify model performance.
- Test the model for edge cases. For example, for sentiment classification, test with sentences with double or triple negation.
- Analyze the mistakes the model is making. The findings from the analysis should be similar to the findings from the analysis of the mistakes it was making during the development phase. For NLP, packages and techniques like TensorFlow Model Analysis [26], Lime [27], Shap [28], and attention networks [5] can give a deeper understanding of what the model is doing deep down. You can see this in action in Figures 11-2 and 11-3. The insights from these during development and production should not change much.
- Another good practice is to build a subsystem that keeps track of key statistics of the features. Since all features are numerical, we can maintain statistics like mean, median, standard deviation, distribution plots, etc. Any deviation in these statistics is a red flag, and we're likely to see the system churning out wrong predictions. The reason could be as simple as a bug in the pipeline or as complex as a covariate shift in the underlying data. Packages like TensorFlow Model

Analysis [26] can track these metrics. Figure 11-4 shows distributions for metrics of various features for a dataset that can be tracked to find covariate shift or bugs.

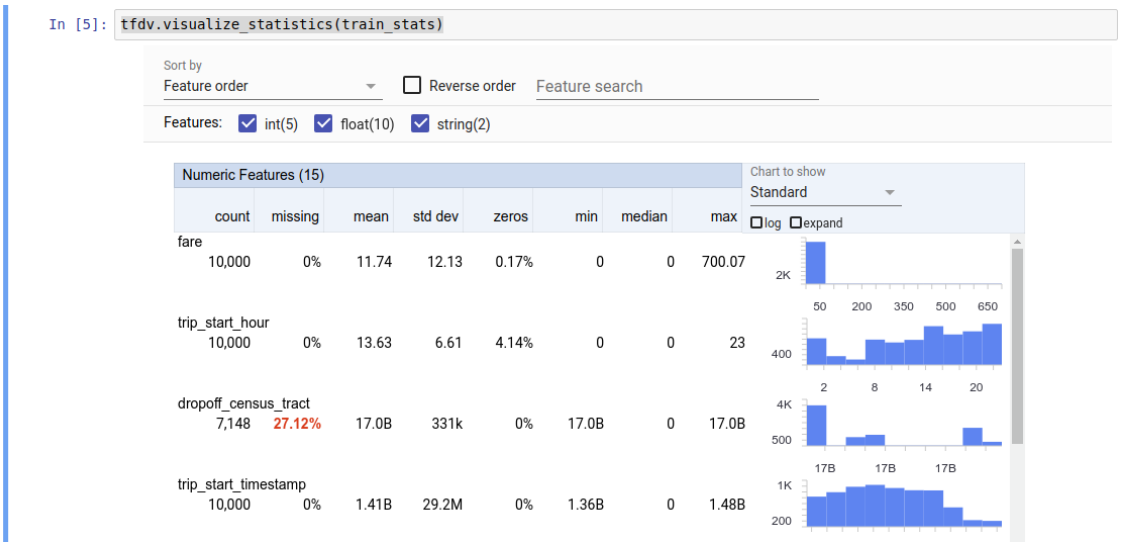


Figure 11-4. Feature statistics in TensorFlow Extended [29]

- Create dashboards for tracking model metrics and create an alerting mechanism on them in case there are any deviations in the metrics. We'll discuss this point in detail in the next section.
- It's always good to know what a model is doing inside. This goes a long way toward understanding why a model is behaving in a certain way. A key question in AI has been how to create intelligent systems where we can explain why the model is doing what it is doing. This is called *interpretability*. It's the degree to which a human can understand the cause of a decision [30]. While many algorithms in machine learning (such as decision trees, random forest, XGboost, etc.) and computer vision have been very interpretable, this is not true for NLP, especially DL algorithms. With recent techniques such as attention networks, Lime, and Shapley, we have greater interpretability in NLP models. Interested readers can look at *Interpretable Machine Learning* by Christoph Molnar [31] for further discussion on this topic.

**Monitoring**

Once an ML system has been deployed and is in production, we need to make sure the model continues working well. As an example deployment, if the model is being trained automatically every day with new data points, certain bugs can creep in, or the model can malfunction. To ensure that this doesn't happen, we need to monitor the model for a range of things and trigger alerts at the right points:

- Model performance has to be monitored regularly. For a web service–based model, it can be the mean and various percentiles—50th (median), 90th, 95th, and 99th (or deeper)—for response time. If the model is deployed as a batch service, statistics on the batch processing and task times have to be monitored.

- Similarly, it helps to store monitor model parameters, behavior, and KPIs. Model KPIs for the abusive comments example would be the percentage of comments that were reported by users but not flagged by the model. For a text classification service, it could be the distribution of classes that are classified each day.
- For all the metrics we're monitoring, we need to periodically run them through an anomaly detection system that can alert changes in normal behavior. This could be a sudden spike in the response rate of a web service or a sudden drop in retraining times. In the worst case, when the performance drops substantially, we may also want to hit circuit breakers (i.e., move to a more stable model or a default approach).
- If our overall engineering pipeline is using a logging framework, there's a good chance it also has support for monitoring anomalies over time for any metric. For instance, ELK stack by Elastic offers built-in anomaly detection [7]. Sumo Logic also flags outliers that can be queried as needed [32]. Microsoft also offers anomaly detection as a service [33].

Monitoring our ML models and their deployments can save substantial time as the project scales. As the system matures and the model stabilizes, proper monitoring allows MLOps teams to largely manage it, so data scientists can solve other harder problems. Although, as systems mature, we also start accumulating more technical debt, which we'll cover in the next section.

**Minimizing Technical Debt**

Throughout this book, and especially in this chapter, we've seen various aspects of training NLP models, deploying them as a part of a larger system, and iteratively improving from there on. As we start iterating from the first version of the system, the system and various components, including the model, can easily become complex. This brings the challenges of maintaining the system. We may have scenarios where we don't know if the incremental improvements justify the complexity. Such scenarios can create a technical debt. Let's take a brief look at addressing technical debt in building AI software.

It's important to plan and build for the future when working with any software system. We have to ensure that our system continues being both performant and easy to maintain after all these continuous iterations and testing. Unused and poorly implemented improvements can create technical debt. If we're not using a feature or any of its combinations with other features, it's important to drop it out of the pipeline. A feature or part of the code that doesn't work just clogs our infrastructure, hinders fast iteration, and brings down clarity.

A good rule of thumb is to look at the coverage a feature provides. If a feature is present in only a few data points, say, 1%, then maybe it's not worth keeping. But even something like this can't be applied blindly. For example, if the same feature covers just 1% of the data but gives 95% classification accuracy just based on that feature, then it's really effective and most certainly worth continuing to use. From our experience, an important tip (that we've also reiterated several times in the book) is: *opt for a simpler model that has performance comparable to a much more complex model if you want to minimize technical debt*. Complex models may become necessary if there's no equivalent simple model though.

Besides these recommendations, we'd also like to share some landmark work on building mature ML systems:

- "A Few Useful Things to Know About Machine Learning" by Pedro Domingoes of the University of Washington [34]
- "Machine Learning: The High-Interest Credit Card of Technical Debt" by a team at Google AI [35]
- "Hidden Technical Debt in Machine Learning Systems" by a team at Google AI [36]
- *Feature Engineering for Machine Learning*, a book written by Alice Zheng and Amanda Casari [22]
- "Ad Click Prediction: A View from the Trenches," a work by a Google Search team on the issues faced by a large online ML system [37]
- "Rules of Machine Learning," an online guide created by Martin Zenkovich of Google [38]
- "The Unreasonable Effectiveness of Data," a report by renowned UC Berkeley researcher Peter Norvig and a Google AI team [39]
- "Revisiting Unreasonable Effectiveness of Data in Deep Learning Era," another modern look at the previous report by a team from Carnegie Mellon University [40]

So far, we've discussed various best practices used in building mature AI systems. From finding better features to version control of datasets, these practices are manual and effort intensive. Driven by the ultimate goal of building intelligent machines and reducing manual effort, an interesting recent work has been to automate some aspects of building AI systems. Let's look at some key efforts in this direction.

**Automating Machine Learning**

One of the holy grails of machine learning is to automate more and more of the feature engineering process. This has led to the creation of a subarea called AutoML (automated machine learning), which aims to make machine learning more accessible. In most cases, it generates a data analysis pipeline that can include data pre-processing, feature selection, and feature engineering methods. This pipeline essentially selects ML methods and parameter settings that are optimized for a specific problem and data. As all of these steps can be time consuming for the ML expert and may be intractable for a beginner, AutoML can be a much-needed bridge for a gap in the world of machine learning. AutoML is itself essentially "doing machine learning using machine learning," making this powerful and complex technology more widely accessible for those hoping to make use of massive amounts of data.

As an example, one research group at Google has used AutoML techniques [41] for language modeling with the Penn Treebank dataset. Penn Treebank is a benchmark dataset for linguistic structure [42]. The research group at Google found that their AutoML approach can design models that achieve accuracies on par with state-of-the-art models designed by world-class machine learning experts. Figure 11-5 shows an example of a neural network generated by AutoML.

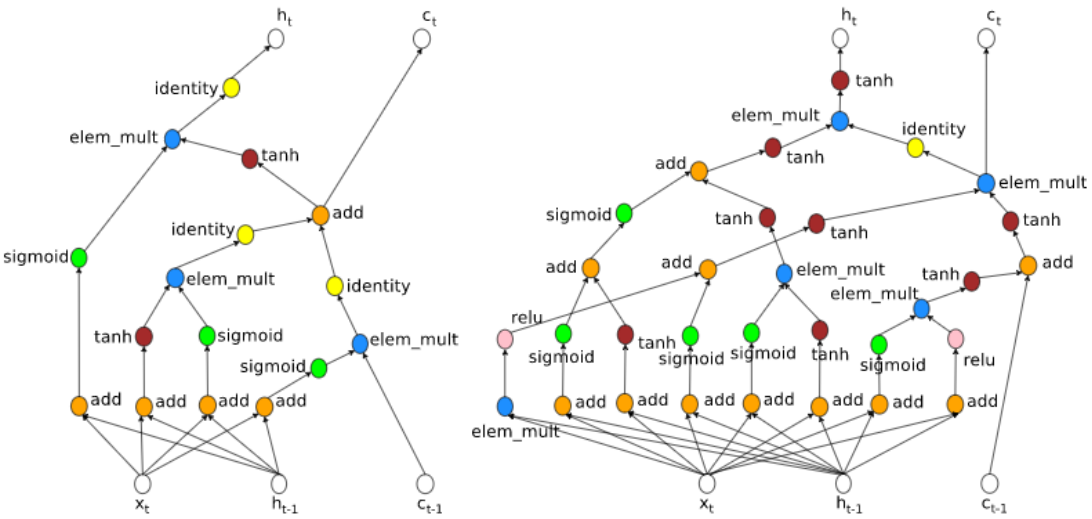


Figure 11-5. AutoML-generated network [41]

On the left side of the figure is a neural network that Google experts created to parse text. On the right side is another network that was created automatically by Google's AutoML. AutoML that explores various neural network architectures automatically performed as well as the handcrafted model. It's fascinating to see that their system did almost as well as humans even for designing ML models.

AutoML is the cutting edge of machine learning. One should only build it from the bottom up when more traditional methods for improving performance are exhausted. It often requires a high amount of computing and GPU resources and a higher level of technical skill when doing it from scratch.

**auto-sklearn**

As we mentioned previously, it's generally a good idea to work on automating machine learning only after most other options have been exhausted. In cases where the need for AutoML [43] is more clear, one of the best libraries for applying it is auto-sklearn. It uses recent advancements in Bayesian optimization and meta-learning to search in a huge hyperparameter space to figure out a reasonably good ML model on its own. As it's integrated with sklearn, which is one of the more popular ML libraries, using it is quite simple:

```
import autosklearn.classification
import sklearn.model_selection
import sklearn.datasets
import sklearn.metrics
X, y = sklearn.datasets.load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = \
    sklearn.model_selection.train_test_split(X, y, random_state=1)
automl = autosklearn.classification.AutoSklearnClassifier()
```

```
automl.fit(X_train, y_train)

y_hat = automl.predict(X_test)

print("Accuracy", sklearn.metrics.accuracy_score(y_test, y_hat))
```

This code builds an autosklearn classifier for the MNIST digits dataset [44]. It splits the dataset into training and test sets. While running for about an hour, this will automatically yield accuracy of over 98%.

When we peek through what's happening internally, we see different stages of AutoML, as shown in the snippet below:

```
[(0.080000, SimpleClassificationPipeline({'balancing:strategy': 'none',

'categorical_encoding:__choice__': 'one_hot_encoding', 'classifier:__choice__':

'lda',

'imputation:strategy': 'mean', 'preprocessor:__choice__': 'polynomial',

'rescaling:__choice__': 'minmax',

'categorical_encoding:one_hot_encoding:use_minimum_fraction': 'True',

'classifier:lda:n_components': 151,

'classifier:lda:shrinkage': 'auto', 'classifier:lda:tol':

0.02939556179271624,

'preprocessor:polynomial:degree': 2, 'preprocessor:polynomial:include_bias':

'True',

'preprocessor:polynomial:interaction_only': 'True',

'categorical_encoding:one_hot_encoding:minimum_fraction': 0.0729529152649298},

dataset_properties={

 'task': 2,

 'sparse': False,

 'multilabel': False,

 'multiclass': True,

 'target_type': 'classification',

 'signed': False})),

...

...

...

...
```

(0.020000, SimpleClassificationPipeline({'balancing:strategy': 'none',

'categorical_encoding:__choice__':

'one_hot_encoding', 'classifier:__choice__': 'passive_aggressive',

'imputation:strategy': 'mean',

'preprocessor:__choice__': 'polynomial', 'rescaling:__choice__': 'minmax',

'categorical_encoding:one_hot_encoding:use_minimum_fraction': 'True',

'classifier:passive_aggressive:C':

0.03485276894122253, 'classifier:passive_aggressive:average': 'True',

'classifier:passive_aggressive:fit_intercept': 'True',

'classifier:passive_aggressive:loss': 'hinge',

'classifier:passive_aggressive:tol': 4.6384320611389e-05,

'preprocessor:polynomial:degree': 3,

'preprocessor:polynomial:include_bias': 'True',

'preprocessor:polynomial:interaction_only': 'True',

'categorical_encoding:one_hot_encoding:minimum_fraction': 0.11994577706637469},

dataset_properties={

 'task': 2,

 'sparse': False,

 'multilabel': False,

 'multiclass': True,

 'target_type': 'classification',

 'signed': False})),

]

auto-sklearn results:

 Dataset name: d74860caaa557f473ce23908ff7ba369

 Metric: accuracy

 Best validation score: 0.991011

 Number of target algorithm runs: 240

 Number of successful target algorithm runs: 226

 Number of crashed target algorithm runs: 1

 Number of target algorithms that exceeded the time limit: 2

Number of target algorithms that exceeded the memory limit: 11

Next, let's take a look at Google Cloud services, as well as a few other approaches to NLP problems.

**Google Cloud AutoML and other techniques**

Google Cloud Services has also recently released AutoML as a service. This doesn't require any technical knowledge beyond providing training data in the expected format. They've specifically built Cloud AutoML services for different parts of AI, including computer vision and structured tabular data, as well as for NLP.

For NLP, their Cloud AutoML is applied automatically when training custom models for:

- Text classification
- Entity extraction
- Sentiment analysis
- Machine translation

For all these tasks, Google Cloud has defined a specific format that the AutoML models expect the data to be in. More information on these can be found in their documentation [45, 46]. Microsoft also has tooling for AutoML in their Azure Machine Learning [47].

Another interesting approach to tackling an NLP problem in a more automated way is to use the AutoCompete framework created by Abhishek Thakur [48], a top-ranked Kaggle Competitions Grandmaster. Even though his initial approach was to focus on any data science problem specifically targeted to competitions, it has now evolved to a general framework to solve such problems. He has also released a detailed notebook titled "Approaching (Almost) Any NLP Problem on Kaggle" [49] that creates a general modeling framework for NLP problems with a well-defined dataset and goals. While this may not completely solve the specific NLP task you're working at, it's a good start to look at creating baseline models.

So far, we've addressed a range of issues that might come up when trying to build, deploy, and maintain NLP software. However, an equally important component of such an endeavor is to follow standard product development processes. While the field of software development processes and life cycle is well established, there are some important things to consider while working on projects that involve predictive models like the ones we've discussed throughout the book. Let's now take a look at that aspect.

The Data Science Process

Data science is a broad term describing the algorithms and processes used to extract meaningful information and actionable insights from all forms of data. Thus, all NLP work in the industry can be categorized under the data science umbrella. While data science as a term is relatively new, it's been around in some form or another for the past few decades. Over the years, people have formulated and formalized the best processes and practices of working with data. Two popular processes in the industry are the KDD process and the Microsoft Team Data Science Process.

**The KDD Process**

The ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD) is one of the oldest and most reputed data mining conferences in the world. Some of the founders of the conference also created the KDD process in 1996. The KDD process [50], depicted in Figure 11-6, consists of a series of steps that should be applied to a data science or data mining problem to get better results.

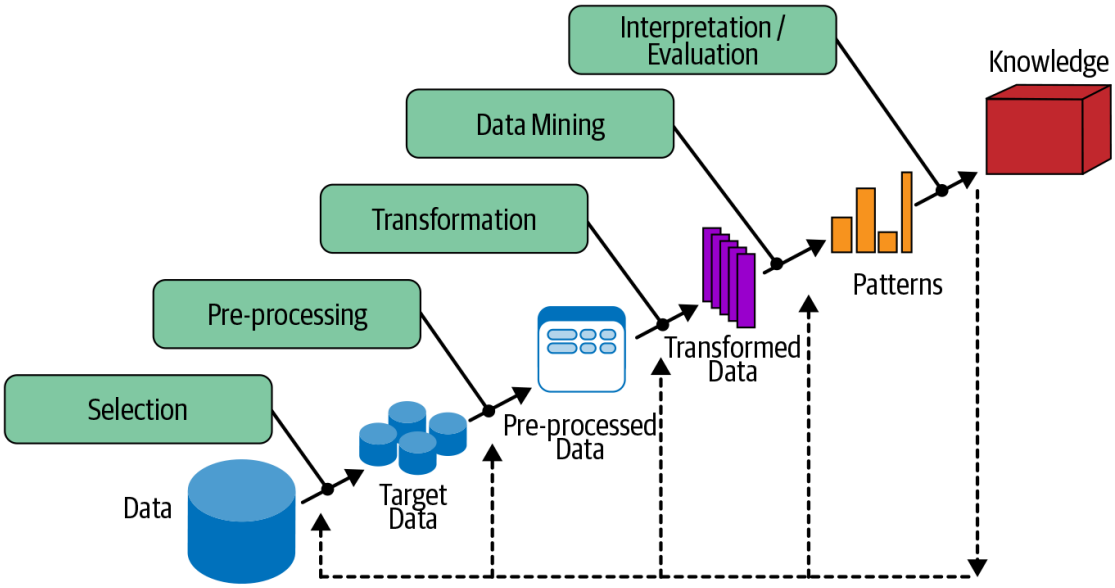


Figure 11-6. The KDD process [50]

These steps are ordered as follows:

1. *Understanding the domain:* This includes learning about the application and understanding the goals of a problem. It also involves getting deeper into the problem domain and extracting relevant domain knowledge.
2. *Target dataset creation:* This includes selecting a subset of data and variables the problem will focus on. We may have a plethora of data sources at our disposal, but we focus on the subset we need to work on.
3. *Data pre-processing:* This encompasses all activities needed so that the data can be treated coherently. This includes filling missing values, noise reduction, and removing outliers.
4. *Data reduction:* If the data has a lot of dimensions, this step can be used to make it easier to work with. This includes steps like dimensionality reduction and projecting the data into another space. This step is optional depending on the data.
5. *Choosing the data mining task:* Various classes of algorithms can be applied to a problem. They may be regression, classification, or clustering. It's important to select the right task based on our understanding from Step 1.
6. *Choosing the data mining algorithm:* Based on the selected data mining task, we need to select the right algorithm. For instance, for classification, we can choose algorithms such as SVM, random forests, CNNs, etc., as we saw in Chapter 4.

7. *Data mining:* This is a core step of applying the selection algorithm from Step 6 to the given dataset and creating predictive models. Tuning with respect to parameters and hyperparameters also happens here.
8. *Interpretation:* Once the algorithm is applied, the user has to interpret the results. This can be done partially by visualizing various components of results.
9. *Consolidation:* This is the final step where we deploy the built model into an existing system, document the approach, and generate reports.

As seen in the figure, the KDD process is highly iterative. There can be any number of loops between various steps. At each step, we can and may need to go back to earlier steps and refine the information there before moving ahead. This process is a good reference when working on a specific data science problem. While not exactly the same, the pipelines we've discussed throughout the book deal with the same idea of bringing structure to building NLP systems. Now, let's take a look at the second process.

**Microsoft Team Data Science Process**

The KDD process was introduced in the late '90s. As the fields of machine learning and data science grew, bigger teams working exclusively on such data science projects began to emerge. Further, in the fast-moving world of data-driven development, more flexible and iteration-based frameworks were needed, so other data science processes began to emerge. The Microsoft Team Data Science Process (TDSP) addresses this. It was released by the Microsoft Azure team in 2017 and is one of the modern processes for applying machine learning and working in data science [51].

TDSP is an agile, iterative data science process for executing and delivering advanced analytics solutions. It's designed to improve the collaboration and efficiency of data science teams in enterprise organizations. The main features of TDSP are:

- A data science life cycle definition
- A standardized project structure, which includes project documentation and reporting templates
- An infrastructure for project execution
- Tools for data science, like version control, data exploration, and modeling

The TDSP documentation [52] provides detailed insight into all of these aspects, so we'll just take a brief look in this section. The TDSP data science life cycle, showing different phases of a data project, is shown in Figure 11-7.

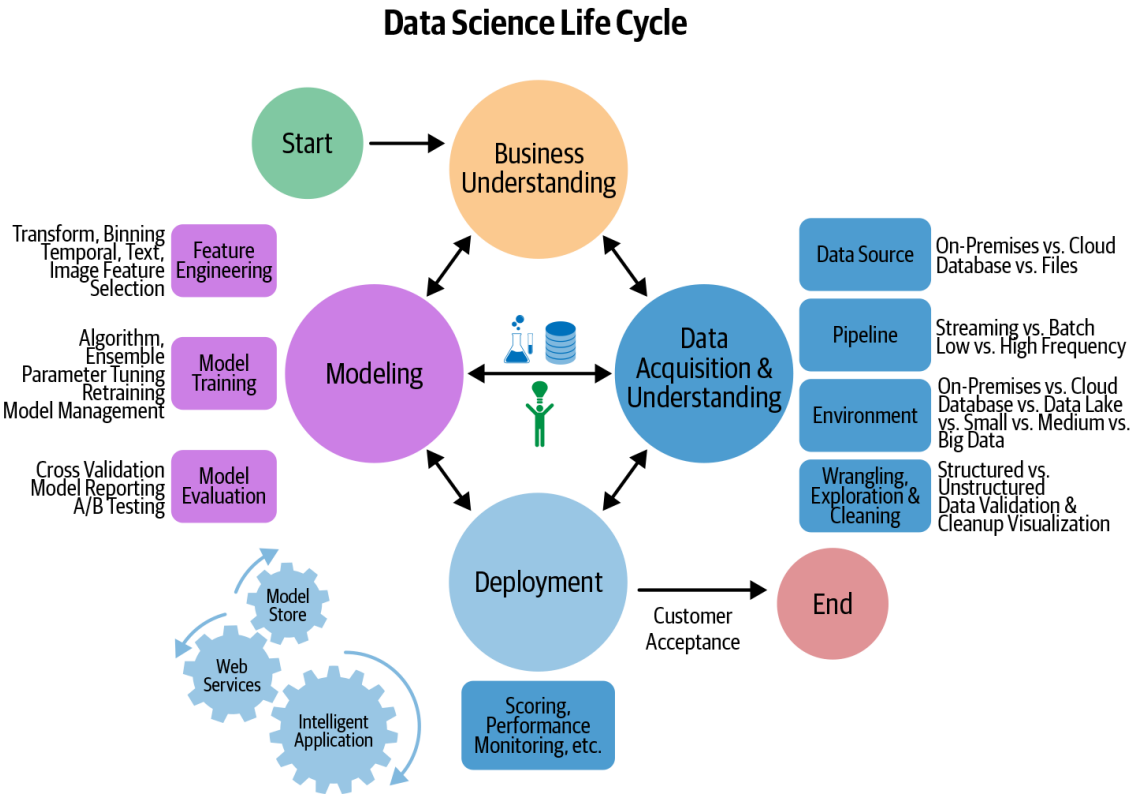


Figure 11-7. The Microsoft TDSP life cycle [51]

While TDSP shares some similarities with the KDD process, an interesting aspect of TDSP is that it defines a life cycle of a data science project from a business and team management perspective. This includes the following stages:

- Business understanding
- Data acquisition and understanding
- Modeling
- Deployment
- Customer acceptance

At a high level, the data science life cycle showcases how various components of an effective and agile data science team should operate. The “Charter” and “Exit Report” documents in the TDSP documentation are particularly important to consider. They help define the project at the start of an engagement and provide a final report to the customer or client.

Overall, these processes can be useful for taking the problems and solutions we’ve discussed so far in this book from prototyping to deployment in a production system. These processes are of course not specific to NLP and are more generic recommendations for any data-driven projects involving ML approaches. While there are other similar project management processes for data science that are emerging as the field grows, we hope this gives you an overview of what to look out for in managing your own NLP projects in a software development setting.

Making AI Succeed at Your Organization

So far, this book has focused on successfully building and deploying solutions for various AI problems. Success of any AI project is dependent not just on the technical superiority of the solution—there are many other factors involved, too. It's a known fact that a large number of AI projects in industry fail because the model doesn't get deployed or, if deployed, fails to achieve its objectives. According to a recent study by Gartner [53], more than 85% of AI projects fail. Here, we discuss some key points and rules of thumb to make AI projects succeed. Many of these points come from our own experience of working in various domains of AI across various organizations.

**Team**

It's important to have the right team to solve the AI problems at hand. In understanding the problem statement, prioritizing, developing, deploying, and consuming, a lot depends on the skills of the team. While there's no fixed recipe, in our experience, the right blend comes with having (1) scientists who build models, (2) engineers who operationalize and maintain models, and (3) leaders who manage AI teams and strategize. It's good to have (4) scientists who have worked in industry after graduate school, (5) engineers who understand scale and data pipelines, and 6) leaders who have also been individual contributor scientists in the past. While (5) is pretty self-explanatory, (4) and (6) warrant some explanation.

Let's look at (4) first. It's important that scientists understand the fundamentals of machine learning and are able to think of novel solutions. Graduate school (especially a PhD) prepares you well for that. But, in industry, solving an AI problem is not just applying novel algorithms. It's also about collecting and cleaning the data, making the data consumption-ready, and applying known techniques. This is very different from academia, where most work happens on known public datasets that are both readily available and clean. Most researchers in academia work on devising novel approaches to beat the state-of-the-art results. In many cases, scientists fresh from academia end up applying sophisticated approaches that prove counterproductive. One is building AI for products—AI is just a means, and not the end. That's why it's important that senior scientists on the team have built and deployed models in industrial settings.

Moving on to (6): AI leadership is very different from software engineering leadership. Even though what runs in production in any AI system is code, AI is fundamentally different from software engineering. Many leaders and organizations are not aware of this nuance. They believe that, because it's code, all the principles of software engineering apply to it. From defining the problem statement to planning project timelines, developing an AI system is different from developing a traditional IT system. This is why it's recommended that AI leaders in your organization have the experience of having been individual contributors (ICs) in the AI field.

**Right Problem and Right Expectations**

In many cases, either the problem at hand is ill defined or AI teams set the wrong expectations. Let's understand this better with some examples. Consider a scenario where we're given a dump of what customers say about a particular product or brand, and we're asked to bring out "interesting" insights. This is a very common scenario in industry; we discussed similar scenarios in "Topic Modeling". Now, can we apply topic modeling to this particular scenario? It depends on what "interesting" means in this context. It could be what the majority of customers are saying, or it could be what a small subset of

customers belonging to a particular region are saying, or it could be what customers are saying about a specific product feature. The possibilities are many. It's important to work with the stakeholders first to clearly define the task. A great way to do this is to take a set of diverse example inputs that include edge cases and ask the stakeholders to write down the desired output. An important thing to keep in mind is that the ready availability of a lot of data does not make something an AI problem by default; many problems can be solved using engineering and rule-based and human-in-the-loop approaches.

Another common problem is stakeholders having wrong expectations of AI technology. This often happens because of articles in popular media that tend to compare AI to the human brain. While that's correct as a motivation behind the area of AI, it's far from the truth. For example, consider a scenario where we built a sentiment analysis system and, for a given input sentence, our system predicts a wrong output. It gives a very high accuracy, but not 100%. Most stakeholders coming from the world of software engineering treat this as a bug and are not willing to accept anything that's not 100% correct. They are not aware of the fact that any AI system (as of today) will give wrong output for a subset of inputs. Another expectation of AI is that it will replace human effort completely, thus saving money. This is seldom the case. It's better to treat AI as augmented intelligence to *support* human efforts rather than artificial intelligence to *replace* human efforts. Also, beyond a point, model performance stagnates and doesn't continue rising with time. We see this in Figure 11-8, where reality behaves more like an S curve while the expectation continues rising.

Even a very mature and advanced AI system requires human supervision. In many cases, we can reduce human efforts, but that happens over a long period of time. In the same vein, stakeholders coming from software engineering may not understand the importance of building responsible AI. Responsible AI ensures trustworthy solutions that are fair, transparent, and accountable. Google [54] and Microsoft [55] have published best practices for building responsible AI systems.

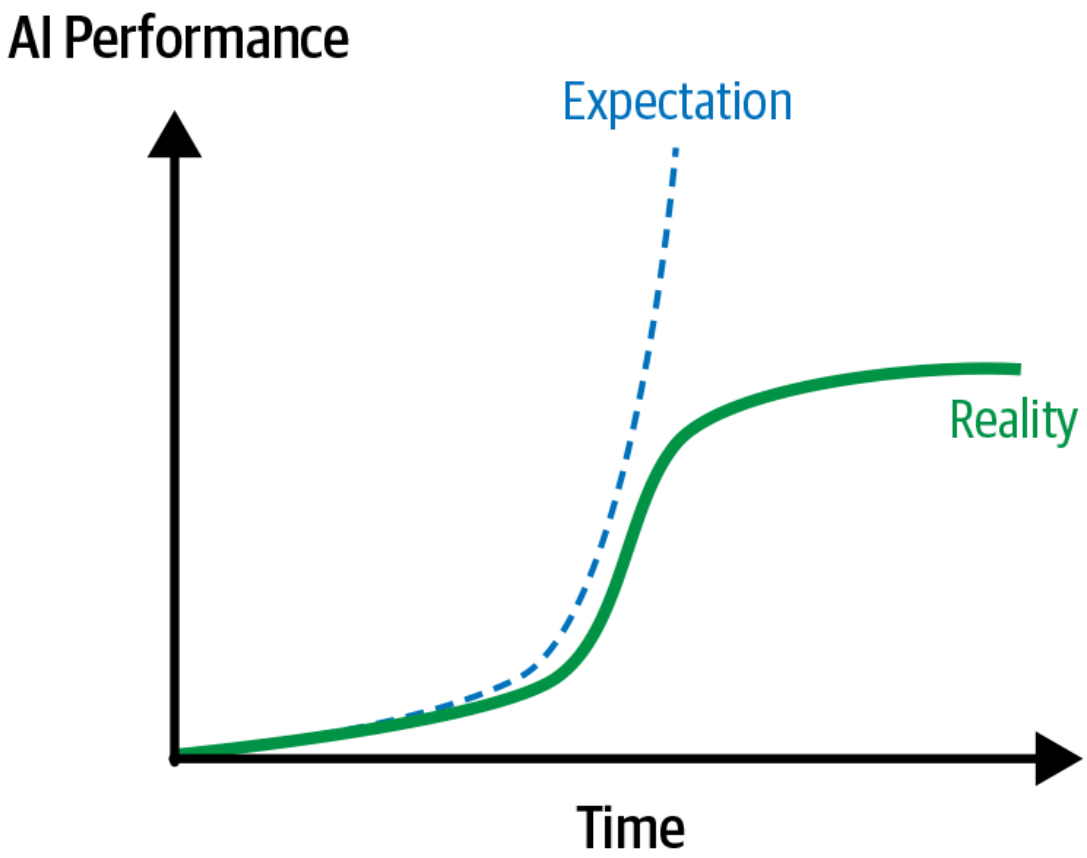


Figure 11-8. Expectation versus reality in AI performance

**Data and Timing**

Data is at the heart of any AI system. We've discussed various aspects of data in detail in previous chapters. Let's look at one more: in many cases, just because an organization has gigabytes or even petabytes of data, it doesn't mean they're ready for AI and can quickly reap its benefits. There's a difference between having data and having the right data. Let's understand this:

*Quality of data*

To perform well, any AI system needs a high quality of data for both training and prediction. What does high quality mean? Data that is structured, homogenous, cleaned, and free of noise and outliers. Going from a dump of noisy data to high-quality data is often a long process. The best way to think of it is the following analogy: raw data is crude oil and AI models are fighter jets. Fighter jets need aviation fuel to fly; they cannot fly on crude oil. So, to enable fighter jets, someone must set up the petroleum refinery to systematically extract the aviation fuel from the crude oil. And setting up this refinery is a long and expensive process.

Another important point is to have the right representative data: data that allows us to solve the problem at hand. For example, there's no way we can improve our search feature if we don't already have the metadata about what we want to search. So, if we don't have "Adidas Shoes Size 10 Tennis Shoes" but only have "Adidas Shoes Size 10," there's no way we can easily make our search help find tennis shoes.

*Quantity of data*

Most AI models are a compressed representation of the dataset used to train them. Not having enough data that's a true representation of the data the model will see in production is a big reason for models not performing well. How much data is enough? This is a hard question to answer, but there are some rules of thumb. For instance, for sentence classification using baseline algorithms such as Naive Bayes or random forest, we've observed that having at least two to three thousand data points per class is a must to be able to build an acceptable classifier.

*Data labeling*

As of today, most success stories of AI in industry have come from supervised AI. As we discussed in initial chapters, it's the subarea where, for each data point, we have the ground truth. For many problems, the ground truth comes from human annotators. This is often a time-consuming and expensive process. In many industrial settings, stakeholders aren't aware of the importance of this step.

Data labeling is often a continuous process. While we do get data labeled in bulk as a one-time effort to build the first versions of our model, once the model is put in production and stabilizes, getting the production data annotated is a continuous process from there on. Further, we need to define processes for labeling and enforce quality checks to improve the accuracy and consistency of human annotators. This is done using metrics like kappa to measure inter-annotator agreement [56].

Currently, AI talent comes at a high cost. Without the right data, it will be futile to hire AI talent; having the right data is a prerequisite for AI teams to deliver well and fast. By this, we don't mean that we must have all of the prerequisites in place before bringing in AI talent. What we mean is that we must be fully aware of other prerequisites, such as the right data, and have realistic expectations in the absence of it.

**A Good Process**

Another important factor that often leads to the failure of AI projects is not following the right process. In this chapter, we've already discussed both the KDD and Microsoft processes. Both of them are great starting points. Here are some other important points to consider when getting started:

*Set up the right metrics*

Most AI projects in industry aim to solve a business problem. In many cases, teams set up AI metrics like precision, recall, etc., as success metrics. But we must also set up the right business metrics along with AI metrics. For example, let's say we're building a text classifier to automatically assign customer complaints to the right customer care teams. The right metric for this is the number of times a complaint is reassigned to another team. A classifier that has a 95% F1 score but leads to many complaints being reassigned multiple times is of no use. Another example of this is a chatbot system that correctly detects intent but has high user drop-off rates. User interaction and drop-off rates provide a complete picture that's missed by using only AI-specific metrics.

*Start simple, establish strong baselines*

AI scientists are often influenced by the latest techniques and recent state-of-the-art (SOTA) models and apply those in their work straight away. Most SOTA techniques are both compute- and data-intensive, which leads to cost and time overruns. The best way is to start with simple approaches and build strong baselines. Many times, a SOTA technique might only give us marginal improvement over a rule-based system! Try multiple simple approaches first before pondering over complex approaches.

*Make it work, make it better*

Building a model is often only 5–10% of most AI projects; the remaining 90% is made up of various steps, ranging from data collection to deployment, testing, maintenance, monitoring, integration, pilot testing, etc. It's always good to build an acceptable model quickly and complete one full project cycle instead of spending a huge amount of time building an amazing model. This helps all stakeholders realize the value proposition of the project.

*Keep shorter turnaround cycles*

Even when solving a standard problem with well-known approaches, we must still apply them to our dataset to see if they work or not. For example, if we're building a sentiment analysis system, it's a well-known fact that Naive Bayes gives very strong baselines. Yet it's very much possible that for our dataset, Naive Bayes might not give good numbers. Building AI systems involves a lot of experiments to figure what works and what doesn't. Hence, it's important to build models quickly and present the results to stakeholders frequently. This helps raise any red flags early and get early feedback.

There are a few other important things to consider, which we'll cover next.

**Other Aspects**

In addition to the various points we've discussed so far, there are some more key points to consider, including compute costs and return on investment. Let's discuss those now:

*Cost of compute*

Many AI models (especially DL-based models) are compute-intensive. Over time, GPUs on the cloud or physical hardware prove to be considerably expensive. Many organizations are known to spend huge amounts on GPU and other cloud services—so much that they have to create parallel projects to reduce these costs.

*Blindly following SOTA*

Practitioners are often keen to apply SOTA models in their work. This often proves to be disastrous. For example, Meena [57], a SOTA chatbot system from Google that gave amazing results, took over 2,048 TPU for 30 days for training. That compute time is worth $1.4M. While Meena has shown some very impressive results, imagine using Meena techniques to build a chatbot for automating customer support that saves $1,000 a day. We would need to run the chatbot for over four years just to break even on the training cost.

*ROI*

AI projects are expensive; various stages, such as data collection, labeling, hiring AI talent, and compute all involve costs. For this reason, it's important to estimate the gains at the

start of the project itself. We must establish the process and clear metrics to measure the returns early on in the project.

*Full automation is hard*

We can never achieve complete automation, at least for any moderately complex AI project—it will continue to require some manual effort. Figure 11-9 represents this in the same S curve we discussed earlier. Levels for complete automation and acceptable performance might change depending on the project, but the broad point will hold true.

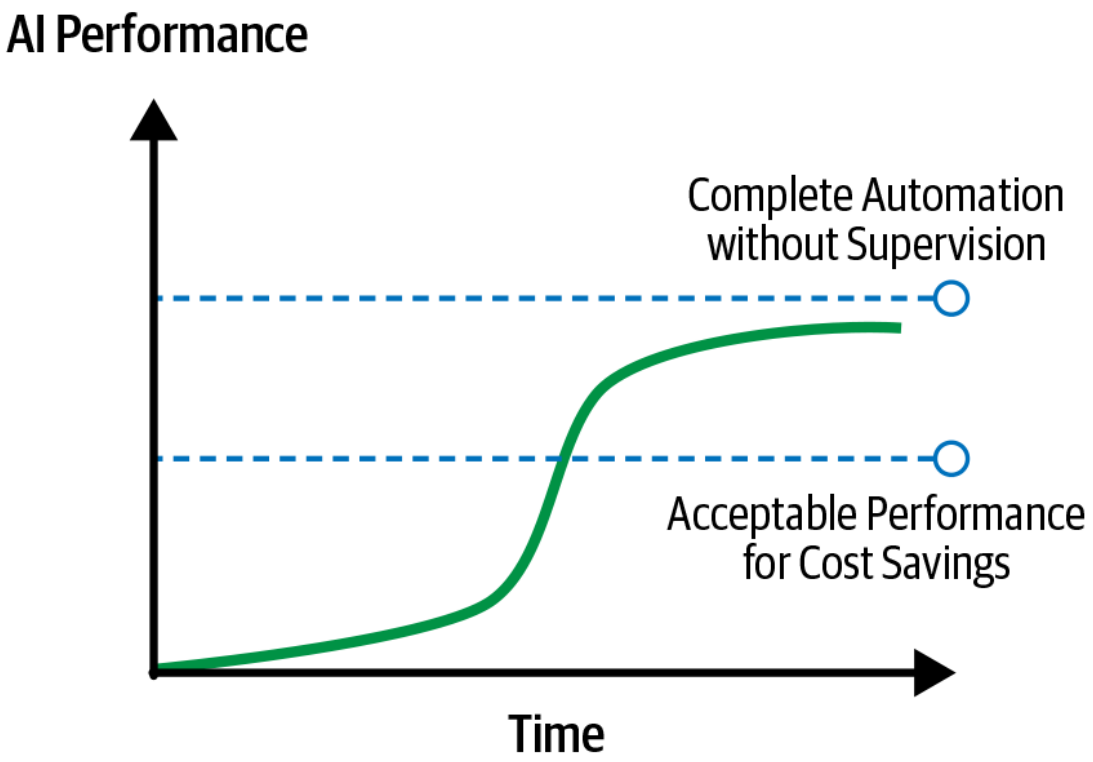


Figure 11-9. Complete automation can be hard

We've covered some key points in this section, but making AI succeed in business is a vast topic. We suggest a few articles for further reading. While some of them bring out the distinctions between software engineering and AI, others discuss rules of thumb for building AI systems:

- "Why Is Machine Learning 'Hard'?," a blog post by S. Zayd Enam, a Stanford researcher [58]
- "Software 2.0," a blog post on AI as a different way of writing software by Andrej Karpathy, a well-known researcher, educator, and scientist at Tesla [59]
- "NLP's Clever Hans Moment Has Arrived," an article by Benjamin Heinzerling that argues the validity of SOTA results obtained on certain popular datasets [60]
- "Closing the AI Accountability Gap," a report by a team at Google AI and the nonprofit Partnership on AI [61]
- "The Twelve Truths of Machine Learning for the Real World," a blog post by Delip Rao, researcher and O'Reilly author [62]

- "What I've Learned Working with 12 Machine Learning Startups," an article by Daniel Shenfeld, a startup veteran and ML consultant [63]

These will give you a more holistic picture. Figure 11-10 demonstrates what we've covered in this section and the chapter.

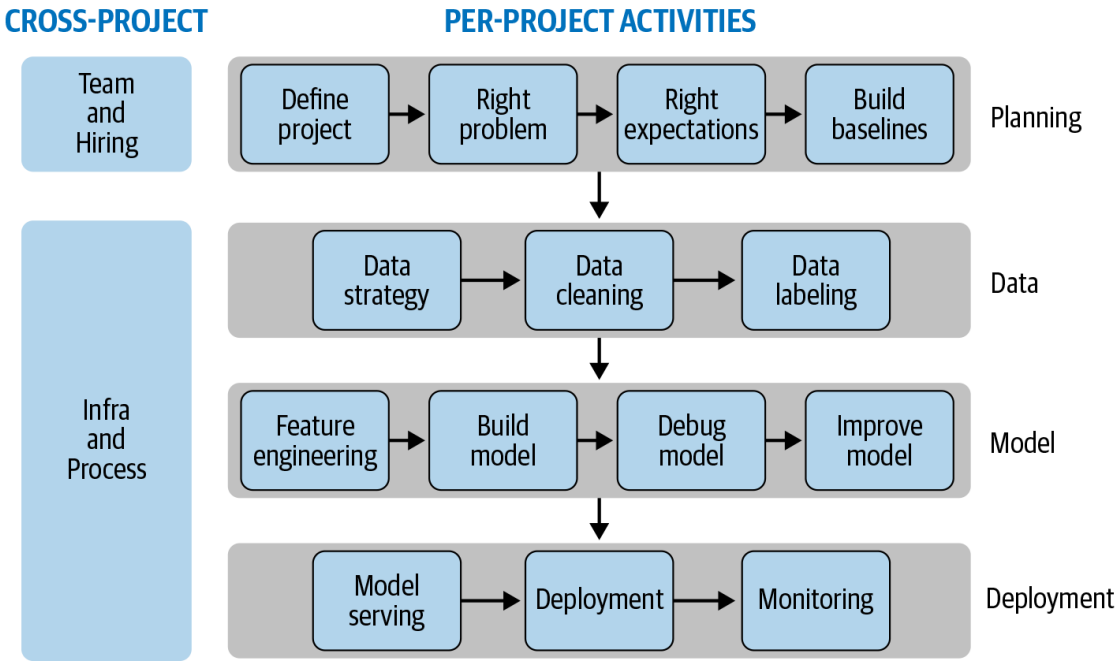


Figure 11-10. Life cycle of an AI project

Many of these suggestions are not hard rules set in stone; their application will depend on the context of your project, problem, data, and organization. We hope the discussion in this section will help in making your AI endeavors succeed.

Peeking over the Horizon

We'd like to end this chapter and the book with various perspectives of how machine learning is evolving. ML will continue improving on the cutting edge, and its applications will be more relevant to business in the coming years. One way to look at this is the influential lecture by renowned scientist C.P. Snow in 1959, titled *The Two Cultures and the Scientific Revolution* [64]. In this lecture, Snow states that the intellectual world can be seen from two distinct perspectives, which seem to be getting more divided over time. One perspective is of science and technology and the other is concerned with arts and humanities. He argues why it's important for these two perspectives to have a common core for better advancements of the entire area. This is true for AI as well.

Analogously, in the world of AI, we see a similar set of two distinct perspectives emerging. On one hand, we have the advances made by researchers and scientists working on the forefront. On the other hand, we have businesses trying to leverage AI. This includes everyone from Fortune 500 companies to early stage startups. The world increasingly believes that the successful adoption of AI in industry will stem from an intersection of both.

From the perspective of researchers and scientists, we see two macro trends: building *truly intelligent machines* and applying *AI for social good*. For instance, François

Chollet of Google has stressed the importance of building better metrics to measure intelligence in “On the Measure of Intelligence” [65]. Most evaluation of AI models at present is inherently narrow in nature and measures specific skills as opposed to broad abilities and general intelligence. Chollet proposes certain measures inspired by testing of human intelligence, including efficiencies in new skill acquisition. They introduce a dataset called Abstraction and Reasoning Corpus (ARC) that’s inspired by a classic IQ test: the Raven’s Progressive Matrices [66]. One such example is presented in Figure 11-11, where the task is for the computer to infer the missing area by looking at the overall input matrix pattern. Work on improving measures of AI is necessary for developing better and more robust AI in the future.

AI and technology in general can be a force for social good. And there are now various initiatives that are working on AI for social good. Wadhwani AI is working on improving maternal and early childhood health with AI [67]. Google AI for Social Good has a range of initiatives, including applying AI to predict and better manage floods [68]. Similarly, Microsoft is using AI to solve global climate issues, improve accessibility, and preserve cultural heritage [69]. Allen AI has been improving common-sense reasoning in NLP through the WinoGrande dataset [70]. Such work by foundations and research labs is helping to incorporate the forefront of ML and NLP to improve human well-being.

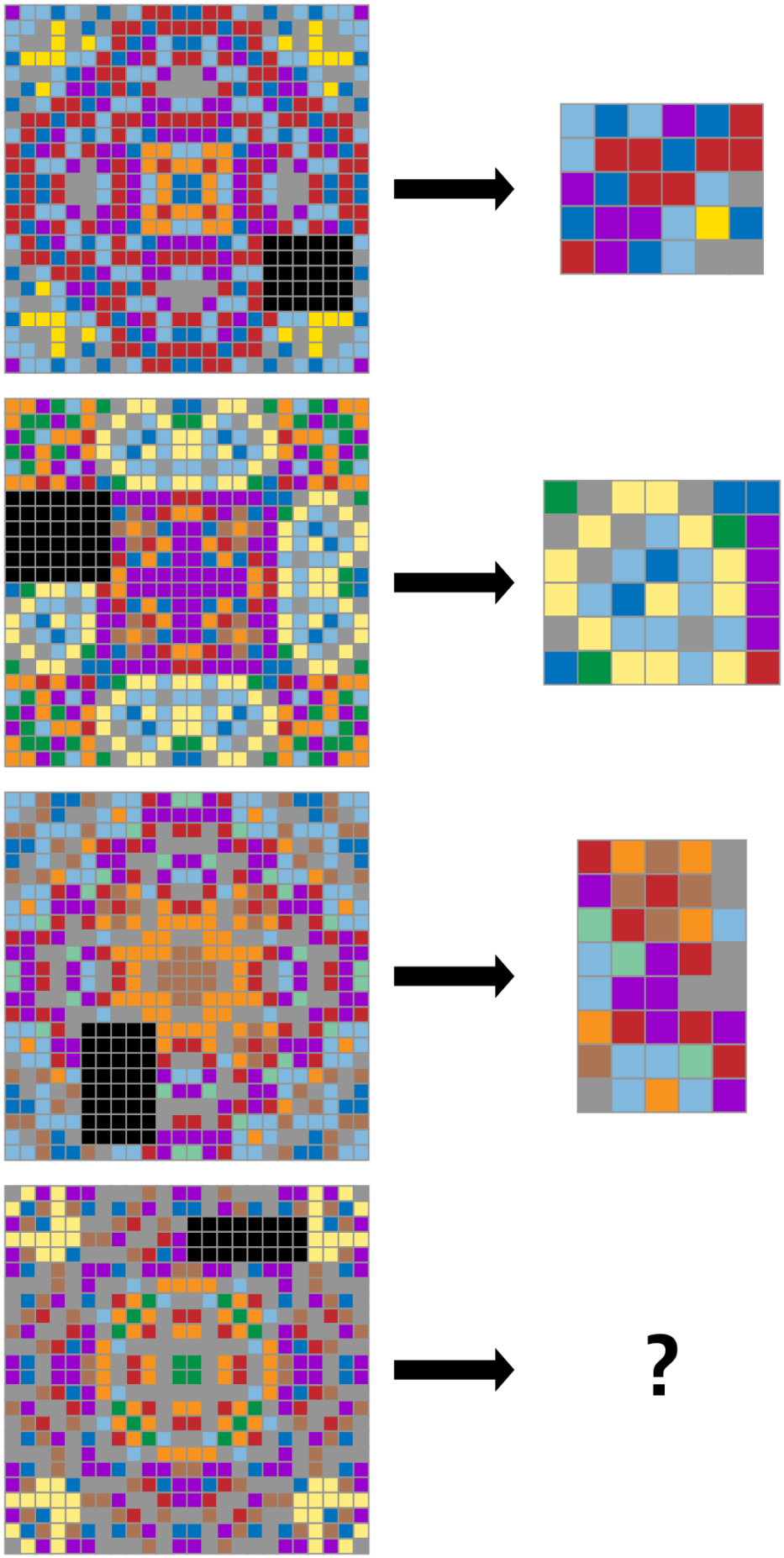

Figure 11-11. Example of an ARC task for general intelligence from [66]

A completely different perspective comes from the world of business. This is more practical and is concerned with business impact and business models. For instance, several consulting firms have conducted surveys across organizations on use cases and effectiveness of AI across industry verticals. McKinsey & Company's Global AI survey is one such example [71]. They discuss how AI has helped different verticals save money by reducing inefficiencies and make more money by expanding the market. They also assess the impact of AI on the workforce and on which parts of organizations it's most impactful. Another such study is a report by MIT Sloan and BCG [72]. This is immensely useful for business leaders to learn how to onboard and grow AI inside their organizations.

Venture capital (VC) firms have been investing heavily in startups building AI-powered businesses. Based on their understanding of how new AI businesses are formed and how they can succeed, they're compiling reports and debriefs. Andressen Horowitz, a major VC firm, has published a report, "The New Business of AI," based on their learnings in many AI investments [73]. The report addresses business issues that AI startups are facing despite the hype, like lower gross margins and product-scaling challenges. They've provided practical advice on building AI businesses that can scale better and be more competitive.

This range of perspectives will be applicable depending on where your organization is in their AI journey. First, when starting a new AI business, lessons from VCs will help you decide what to build. Second, to formulate an AI strategy in a large organization, surveys and reports from industry will align you better. Last but not least, as your organization matures, incorporating SOTA techniques can lead to a step change in your products.

Final Words

And here we come to the end of *Practical Natural Language Processing*! We hope you've learned a few things about NLP tasks and pipelines and their applications in various domains and that these will help you in your day-to-day work. The advances in NLP are just starting to bear big fruits. Some of the most fundamental questions in NLP, like context and common sense, have probably yet to even be asked properly.

True mastery of any skill requires a lifetime of learning, and we hope that our references, research papers, and industry reports will help you continue the journey.

**Footnotes**

**References**

[1] ONNX: An open format built to represent machine learning models. Last accessed June 15, 2020.

[2] Apache Airflow. Last accessed June 15, 2020.

[3] Apache Oozie. Last accessed June 15, 2020.

[4] Chef. Last accessed June 15, 2020.

[5] Microsoft. "MLOps examples". Last accessed June 15, 2020.

[6] Microsoft. MLOps using Azure ML Services and Azure DevOps, (GitHub repo). Last accessed June 15, 2020.

[7] Elastic. "Anomaly Detection".

[8] Krzus, Matt and and Jason Berkowitz. "Text Classification with Gluon on Amazon SageMaker and AWS Batch". *AWS Machine Learning Blog*, March 20, 2018.

[9] The Pallets Projects. "Flask". Last accessed June 15, 2020.

[10] The Falcon Web Framework. Last accessed June 15, 2020.

[11] Django: The web framework for perfectionists with deadlines. Last accessed June 15, 2020.

[12] Docker. Last accessed June 15, 2020.

[13] Kubernetes: Production-Grade Container Orchestration. Last accessed June 15, 2020.

[14] Amazon. AWS SageMaker. Last accessed June 15, 2020.

[15] Microsoft. Azure Cognitive Services. Last accessed June 15, 2020.

[16] Sucik, Sam. "Compressing BERT for Faster Prediction". *Rasa (blog)*, August 8, 2019.

[17] Cheng, Yu, Duo Wang, Pan Zhou, and Tao Zhang. "A Survey of Model Compression and Acceleration for Deep Neural Networks." 2017.

[18] Joulin, Armand, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. "FastText.zip: Compressing Text Classification Models", 2016.

[19] Chee, Cedric. Awesome machine learning model compression research papers, tools, and learning material, (GitHub repo). Last accessed June 15, 2020.

[20] Burkov, Andriy. *Machine Learning Engineering* (Draft). 2019.

[21] Cheng, Heng-Tze. "Wide & Deep Learning: Better Together with TensorFlow." *Google AI Blog*, June 29, 2016.

[22] Zheng, Alice and Amanda Casari. *Feature Engineering for Machine Learning*. Boston: O'Reilly, 2018. ISBN: 978-9-35213-711-4

[23] DVC: Open source version control system for machine learning projects. Last accessed June 15, 2020.

[24] Gundersen, Odd Erik and Sigbjørn Kjensmo. "State of the Art: Reproducibility in Artificial Intelligence." *The Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[25] Gibney, E. "This AI Researcher Is Trying to Ward Off a Reproducibility Crisis." *Nature* 577.7788 (2020): 14.

[26] TensorFlow. "Getting Started with TensorFlow Model Analysis". Last accessed June 15, 2020.

[27] Marco Tulio Correia Ribeiro. Lime: Explaining the predictions of any machine learning classifier, (GitHub repo). Last accessed June 15, 2020.

[28] Lundberg, Scott. Shap: A game theoretic approach to explain the output of any machine learning model, (GitHub repo). Last accessed June 15, 2020.

[29] TensorFlow. "Get started with TensorFlow Data Validation". Last accessed June 15, 2020.

[30] Miller, Tim. "Explanation in Artificial Intelligence: Insights from the Social Sciences", (2017).

[31] Molnar, Christoph. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. 2019.

[32] Sumo Logic. "Outlier". Last accessed June 15, 2020.

[33] Microsoft. "Anomaly Detector API Documentation". Last accessed June 15, 2020.

[34] Domingos, Pedro. "A Few Useful Things to Know about Machine Learning." *Communications of the ACM* 55.10(2012): 78–87.

[35] Sculley, D., Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. "Machine Learning: The High Interest Credit Card of Technical Debt." *SE4ML: Software Engineering for Machine Learning* (NIPS 2014 Workshop).

[36] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, VinayChaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. "Hidden Technical Debt in Machine Learning Systems." *Proceedings of the 28th International Conference on Neural Information Processing Systems* 2 (2015): 2503–2511.

[37] McMahan, H. Brendan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie et al. "Ad Click Prediction: A View from the Trenches." *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2013): 1222–1230.

[38] Zinkevich, Martin. "Rules of Machine Learning: Best Practices for ML Engineering". *Google Machine Learning*. Last accessed June 15, 2020.

[39] Halevy, Alon, Peter Norvig, and Fernando Pereira. "The Unreasonable Effectiveness of Data." *IEEE Intelligent Systems* 24.2 (2009): 8–12.

[40] Sun, Chen, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. "Revisiting Unreasonable Effectiveness of Data in Deep Learning Era." *Proceedings of the IEEE International Conference on Computer Vision* (2017): 843–852.

[41] Petrov, Slav. "Announcing SyntaxNet: The World's Most Accurate Parser Goes Open Source". *Google AI Blog*, May 12, 2016.

[42] Marcus, Mitchell, Beatrice Santorini, and Mary Ann Marcinkiewicz. "Building a Large Annotated Corpus of English: The Penn Treebank". *Computational Linguistics* 19, Number 2, Special Issue on Using Large Corpora: II (June 1993).

[43] Feurer, Matthias, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. "Efficient and Robust Automated Machine Learning." *Advances in Neural Information Processing Systems* 28 (2015): 2962–2970.

[44] Le Cun, Yann, Corinna Cortes and Christopher J.C. Burges. "The MNIST database of handwritten digits". Last accessed June 15, 2020.

[45] Google Cloud. "Features and capabilities of AutoML Natural Language". Last accessed June 15, 2020.

[46] Google Cloud. "AutoML Translation". Last accessed June 15, 2020.

[47] Microsoft Azure. "What is automated machine learning (AutoML)?", February 28, 2020.

[48] Thakur, Abhishek and Artus Krohn-Grimberghe. "AutoCompete: A Framework for Machine Learning Competition", (2015).

[49] Thakur, Abhishek. "Approaching (Almost) Any NLP Problem on Kaggle". Last accessed June 15, 2020.

[50] Fayyad, Usama, Gregory Piatetsky-Shapiro, and Padhraic Smyth. "The KDD Process for Extracting Useful Knowledge from Volumes of Data." *Communications of the ACM* 39.11 (1996): 27–34.

[51] Microsoft Azure. "What is the Team Data Science Process?", January 10, 2020.

[52] Microsoft. "Team Data Science Process Documentation". Last accessed June 15, 2020.

[53] Kidd, Chrissy. "Why Does Gartner Predict up to 85% of AI Projects Will 'Not Deliver' for CIOs?", *BMC Machine Learning & Big Data Blog*, December 18, 2018.

[54] Google AI. "Responsible AI Practices". Last accessed June 15, 2020.

[55] Microsoft. "Microsoft AI principles". Last accessed June 15, 2020.

[56] Artstein, Ron and Massimo Poesio. "Inter-Coder Agreement for Computational Linguistics." *Computational Linguistics* 34.4 (2008): 555–596.

[57] Adiwardana, Daniel and Thang Luong. "Towards a Conversational Agent that Can Chat About...Anything". *Google AI Blog*, January 28, 2020.

[58] Enam, S. Zayd. "Why is Machine Learning 'Hard'?", *Zayd's Blog*, November 10, 2016.

[59] Karpathy, Andrej. "Software 2.0". *Medium Programming*, November 11, 2017.

[60] Heinzerling, Benjamin. "NLP's Clever Hans Moment has Arrived". *The Gradient*, August 26, 2019.

[61] Raji, Inioluwa Deborah, Andrew Smart, Rebecca N. White, Margaret Mitchell, Timnit Gebru, Ben Hutchinson, Jamila Smith-Loud, Daniel Theron, and Parker Barnes. "Closing the AI Accountability Gap: Defining an End-to-End Framework for Internal Algorithmic Auditing", (2020).

[62] Rao, Delip. "The Twelve Truths of Machine Learning for the Real World". *Delip Rao (blog)*, December 25, 2019.

[63] Shenfeld, David. "What I've Learned Working with 12 Machine Learning Startups". *Towards Data Science (blog)*, May 6, 2019.

[64] Snow, Charles Percy. *The Two Cultures and the Scientific Revolution*. Connecticut: Martino Fine Books, 2013.

[65] Chollet, François. "On The Measure of Intelligence", (2019).

[66] John, Raven J. "Raven Progressive Matrices," in *Handbook of Nonverbal Assessment*, Boston: Springer, 2003.

[67] Wadhwani AI. "Maternal, Newborn, and Child Health". Last accessed June 15, 2020.

[68] Matias, Yossi. "Keeping People Safe with AI-Enabled Flood Forecasting". *Google The Keyword (blog)*, September 24, 2018.

[69] Microsoft. "AI for Good". Last accessed June 15, 2020.

[70] Sakaguchi, Keisuke, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. "WinoGrande: An Adversarial Winograd Schema Challenge at Scale", (2019).

[71] Cam, Arif, Michael Chui, and Bryce Hall. "Global AI Survey: AI Proves Its Worth, but Few Scale Impact". *McKinsey & Company Featured Insights*, November 2019.

[72] Ransbotham, Sam, Philipp Gerbert, Martin Reeves, David Kiron, and Michael Spira. "Artificial Intelligence in Business Gets Real." *MIT Sloan Management Review* (September 2018).

[73] Casado, Martin and Matt Bornstein. "The New Business of AI (and How It's Different From Traditional Software)". *Andreesen Horowitz*, February 16, 2020.