1. The Machine Learning Landscape

- Optical Character Recognition (OCR)
- spam filter

What Is Machine Learning?

- the science (and art) of programming computers so they can learn from data
- general definition
    - field of study that gives computers the ability to learn without being explicitly programmed
- engineering-oriented definition
    - a computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E

Why Use Machine Learning?

- automatic adaptation to modified input without the need for manual reprogramming
- good performance in areas where the solution is not strictly defined
- good for problems that are too complex for traditional approaches or have no known algorithm
- discovery of patterns in data mining of large data sets that were not immediately apparent

Types of Machine Learning Systems

- criteria
    - trained under human supervision
        - supervised
        - unsupervised
        - semisupervised
        - Reinforcement Learning
    - learn incrementally or on the fly
        - online
        - batch learning
    - compare new data points to known data points or detect patterns and build predictive model
        - instance-based learning
        - model-based learning

Supervised/Unsupervised Learning

- supervised
    - training data fed to the algorithm includes the desired solutions
    - labels - desired solutions
    - classification - train with many examples and model learns how to classify
    - regression - predict a target value when given a set of features called predictors
    - important algorithms
        - k-nearest neighbors
        - linear regression
        - logistic regression
        - support vector machines (SVMs)
        - decision trees and random forests
        - neural networks
- unsupervised learning
    - training data is unlabeled
    - important algorithms
        - clustering
            - k-means
            - hierarchical cluster analysis (HCA)
            - expectation maximization
        - visualization and dimensionality reduction
            - principal component analysis (PCA)
            - kernel PCA
            - locally-linear embedding (LLE)
            - t-distributed stochastic neighbor embedding (t-SNE)
        - association rule learning
            - apriori
            - eclat
    - clustering - detect groups of similarities
    - hierarchical clustering - subdivides detected groups into subgroups
    - visualization algorithms are a form of unsupervised learning
    - dimensionality reduction - simplify data without losing too much information
    - feature extraction
    - TIP: often good to perform dimensionality reduction on data before feeding it to another ML algorithm
    - anomaly detection
    - association rule learning - discover interesting relationships between attributes
- semisupervised learning
    - some partially labeled and some unlabeled data

    - deep belief networks (DBNs)
        - restricted Boltzmann machines (RBMs)
- reinforcement learning
    - learning system (agent) observes environment and performs actions
        - rewards or penalties are given for actions
        - system learns by itself to select a policy (or strategy)

Batch and Online Learning

- batch
    - system is incapable of learning incrementally
    - must be trained using the full data set, generally offline
    - offline learning - trained and then launched to production
- online learning
    - train system incrementally either by feeding it data instances sequentially or in small groups called mini-batches
    - good for systems receiving a continuous flow of data
    - also good for datasets that cannot fit in one machine's memory
    - online learning is generally done offline! (think of it as incremental learning)
    - learning rate - important parameter - how fast system should adapt to changing data
        - high learning rate - system will rapidly adapt to new data, but it will also tend to quickly forget the old data (you don’t want a spam filter to flag only the latest kinds of spam it was shown).
        - low learning rate - system will learn more slowly (more inertia), but it will also be less sensitive to noise in the new data or to sequences of nonrepresentative data points
    - challenge with online learning - if bad data is fed to the system, the system’s performance will gradually decline

Instance-Based Versus Model-Based Learning

- how the system generalizes
- instance-based learning
    - system learns by recording information then generalizes to new cases using a similarity measure
- model-based learning
    - build a model from the examples and then use model to make predictions
    - noisy data
    - model selection
    - linear model
    - model parameters
    - utility/fitness function
    - linear regression and training the model

    - process
        - study data
        - select model
        - train model on data (learning algorithm searched the model parameter values that minimize a cost function)
        - apply model to make predictions on new cases (inference)

Main Challenges of Machine Learning

- bad algorithm (poor algorithm selection)
- bad data
- insufficient quality of training data
    - typically need a large amount of data
    - famous paper showed that vary different (and even vary simple) ML algorithms perform almost identically well for complex problems (natural language) when given enough data
- nonrepresentative training data
    - need data that is representative of the new instances that you want to generalize to for predictions
    - small sample results in sampling noirs
    - sampling bias
- poor quality data
- irrelevant features
    - feature engineering
        - feature selection - selecting the most useful features to train on among existing features
        - feature extraction - combining existing features to produce a more useful one (dimensionality reduction algorithms can help)
        - creating new features by gathering new data
- overfitting the training data
    - overgeneralization
    - overfitting happens when the model is too complex relative to the amount and noisiness of the training data
        - solutions
            - simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data or by constraining the model
            - gather more training data
            - reduce the noise in the training data (e.g., fix data errors and remove outliers)
    - regularization - constraining a model to make it simpler and reduce the risk of overfitting
- underfitting the training data
    - solutions
        - select a more powerful model, with more parameters

        - feed better features to the learning algorithm (feature engineering)
        - reduce the constraints on the model (e.g., reducing the regularization hyperparameter)
- overview
    - machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules
    - there are many different types of ML systems
        - supervised vs unsupervised
        - batch vs online
        - instance-based vs model-based
    - in an ML project you gather data in a training set, and you feed the training set to a learning algorithm
    - if the algorithm is model-based it tunes some parameters to fit the model to the training set (i.e., to make good predictions on the training set itself), and then hopefully it will be able to make good predictions on new cases as well
    - if the algorithm is instance-based, it just learns the examples by heart and uses a similarity measure to generalize to new instances
    - the system will not perform well if training set is too small, or if the data is not representative, noisy, or polluted with irrelevant features (garbage in, garbage out)
    - the model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit)

Testing and Validating

- generally not great to put model into production then observe performance
- split data
    - training set - %80
    - test set - %20
- generalization error (out-of-sample error)
- train model on test set to get an estimate of generalization error
- if the training error is low (i.e., your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data
- if a model generalizes better but needs regularization
    - for regularization hyperparemeter
        - train 100 models using 100 values
            - problem - measured the generalization error multiple times on the test set, and adapted the model and hyperparameters to produce the best model for that set
            - model is unlikely to perform as well on new data
        - use a validation set
            - train multiple models with various hyperparameters using the training set
            - select the model and hyperparameters that perform best on the validation set
            - when model seems to perform well, run a single final test against the test set to get an estimate of the generalization error

    - use cross-validation to avoid wasting too much training data in validation sets
    - cross-validation
        - the training set is split into complementary subsets, and each model is trained against a different combination of these subsets and validated against the remaining parts
        - once the model type and hyperparameters have been selected, a final model is trained using these hyperparameters on the full training set, and the generalized error is measured on the test set

2. End-toEnd Machine Learning Project

- process summary
    - get an overview of project
    - retrieve data
    - discover and visualize the data to gain insights
    - prepare the data for ML algorithms
    - select a model and train it
    - fine-tune model
    - present solution
    - launch, monitor, and maintain system

Working with Real Data

- popular open data repositories
    - UC Irvine Machine Learning Repository
    - Kaggle datasets
    - Amazon's AWS datasets
- meta portals (they list open data repositories):
    - http://dataportals.org/
    - http://opendatamonitor.eu/
    - http://quandl.com/
- ther pages listing many popular open data repositories
    - Wikipedia's list of Machine Learning datasets
    - Quora.com question
    - Datasets subreddit

Look at Big Picture

- see ML project checklist in appendix B
- frame the problem
    - business objectives etc.

- select a performance measure
    - root mean square (RMSE)
    - mean absolute error (MAE)
    - norms
        - Euclidean norm
        - Manhattan norm
- check the assumptions

Get the Data

- create the workspace
    - Jupyter
    - virtualenv
- download the data
    - script it (urllib)
- observe structure of data
- create a test set

Discover and Visualize the Data to Gain Insights

- matplotlib, Pandas, D3.js, etc

Prepare the Data for Machine Learning Algorithms

- TODO: more here
- write functions to prepare data for reuse and automation
- data cleaning
- handling text and categorical attributes
- custom transformers
- feature scaling
- transformation pipelines

Select and Train a Model

- TODO: more here
- training and evaluating on the training set
- better evaluation using cross-validation

Fine Tune Your Model

- grid search
- randomized search

- ensemble methods
- analyze the best models and their errors
- evaluate system on the test set

Launch, Monitor, and Maintain Your System

3. Classification

MNIST

- 70,000 small images of handwritten digits
- commonly used for testing
- sklearn datasets
    - DESCR - description of dataset
    - data - array of one row per record and one column per feature
    - target - key containing array with labels

Training a Binary Classifier

- binary classifier - capable of distinguishing between just two classes
- stochastic gradient descent (SGD)
    - stochastic - randomly determined, having a random probability distribution
    - good starting point for binary classifier
    - capable of handling large datasets efficiently
    - trains instances independently
    - well suited for online learning

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state = 42)
sgd_clf.fit(X_train, y_train_5)
sgd_clf.predict([some_digit])
```

Performance Measures

- classifiers are often more difficult to evaluate than regressors

Measuring Accuracy Using Cross-Validation

- may need more control than sklearn's built-in cross-validation support

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base
import clone

skfolds = StratifiedKFold(n_splits = 3, random_state = 42)
for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = (y_train_5[train_index])
    X_test_fold = X_train[test_index]
    y_test_fold = (y_train_5[test_index])
    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))  # prints 0.9502, 0.96565 and 0.96495
```

- k-fold cross-validation - split training set into k-folds then make predictions and evaluate on each fold using a model trained on the remaining folds
- be aware that accuracy is generally not the preferred performance measure for classifiers
    - especially not for skewed datasets (some classes are more frequent than others)

Confusion Matrix

- counts of the number of times class A was confused and classified as class B (with counts written to the corresponding matrix cell)
    - each row represents and actual class
    - each column represents a predicted class
    - negative class, true positives, true negatives, false positives, false negatives
- precision of classifier - accuracy of positive predictions

```
precision = TP / (TP + FP)
TP - true positives
FP - false positives
```

- recall (sensitivity or true positive rate (TPR)) - ratio of positive instances that are correctly detected by the classifier

```
recall = TP / (TP + FN)
FN - false negatives
```

Precision and Recall

- sklearn provides many classifier metrics functions (including precision and recall)

```
from sklearn.metrics import precision_score, recall_score
```

- F1 score - harmonic mean of precision and recall
  - whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values
  - the classifier will only get a high F1 score if both recall and precision are high

```
F1 = 2 / ((1 / precision) + (1 / recall)) = 2 * (precision * recall) / (precision
```

- importing

```
from sklearn.metrics import precision_score, recall_score
```

- precision/recall tradeoff - increasing precision reduces recall and vice versa

Precision/Recall Tradeoff

- decision function -
- to make classification decisions SGD classifier computes a score based on the decision function and if that score is greater than a threshold, it assigns the instance to the positive class, or else it assigns it to the negative class
- decision threshold
- sklearn does not allow setting threshold directly but can access the value
- can plot precision and recall as functions of the threshold value using matplotlib
- increasing precision requires setting a recall

The ROC Curve

- receiver operating characteristics (ROC) - plots true positive rate (another name for recall) against the false positive rate
  - FPR - ratio of negative instances that are incorrectly classified as positive (= 1 - TNR)
  - TNR (specificity) - ratio of negative instances that are correctly classified as negative
  - ROC = sensitivity (recall) vs. 1 - specificity
- similar to precision/recall curve
- as a rule of thumb, prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives, and the ROC curve otherwise

Multiclass Classification

- aka multinomial classifiers - used to distinguish between more than two classes
- can handle multiples classes directly
  - random forest classifiers

  - naive Bayes classifiers
  - - more
- only binary
  - support vector machine classifiers
  - linear classifiers
- can use various strategies to perform multiclass classification using binary classifiers
- one-versus-all (OvA) strategy (one-versuse-the-rest)
- one-versus-one (OvO)
- sklearn auto-detects when a binary classifier is used for multiclass classification and runs OvA (except for SVM which uses OvO)
- WARNING: when a classifier is trained, it stores the list of target classes in its classes_ attribute, ordered by value

```
from sklearn.multiclass import OneVsOneClassifier
```

Error Analysis

- to improve models, first analyze the types of errors that they make
- check confusion matrix
  - often easier to view as an image
- with MNIST, can process the images
  - Scikit-Image
  - Pillow
  - OpenCV
- analyzing individual errors can be useful but time consuming

Multilabel Classification

- trained on multiple labels and outputs set of binary values for each label
- there are many ways to evaluate a multilabel classifier, and selecting the right metric depends on the project
  - one approach is to measure the F1 score for each individual label (or any other binary classifier metric discussed earlier), then simply compute the average score

Multioutput Classification

- multioutput-multiclass classification (or multioutput classification) - a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values)
- NOTE: The line between classification and regression is sometimes blurry
  - predicting pixel intensity is more akin to regression than to classification
  - multioutput systems are not limited to classification tasks

    - can have a system that outputs multiple labels per instance, including both class labels and value labels

4. Training Models

- can do a lot without knowing what is under the hood
    - optimized a regression system
    - improved a digit image classifier
    - built a spam classifier from scratch

Linear Regression

- linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the bias term (also called the intercept term)
- linear regression model prediction

```
^y = O0 + O1x1 + ... Onxn
^y - predicted value
n - number of features
xi - ith feature value
0j - jth model parameter (including bias term O0 and feature weights O1 ... On)
```

- linear regression model - vectorized form

```
^y = h0(x) = O^T * x   // * = dot
O - model's parameter vector containing the bias term O0 and the feature weights O
x - instance's feature vector containing x0 to xn, with x0 always 1
O^T*x - dot product of O^T and x
h0 - hypothesis function using model parameters O
```

- training a model means setting its parameters so that the model best fits the training set
- to measure performance, find the values of the parameters that results in a minimal RMSE (root mean square error)
- in practice it is simpler to minimize the MSE (mean square error) and it leads to the same result

```
MSE(X, h0) = 1 / m Sum from i = 1 to m (O^T * x^(i) - y ^(i))^2
```

- the normal equation
    - used to find the value of θ that minimizes the cost function in a closed-form solution
    - e.g. an equation that gives the result directly

```
^O = (X^T * X)^-1 * X^T * y
^O - value O that minimizes the cost function
y - vector of target values containing y1 to ym
```

- np.linalg -> inv() (inverse of matrix) and dot() (dot product)
- using y = 4 + 3x + Gaussian noise

```
# using numpy
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

# using sklearn
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
# (array([ 4.21509616]), array([[ 2.77011339]]))
```

- the Normal Equation computes the inverse of XT · X, which is an n × n matrix (where n is the number of features)
- the computational complexity of inverting such a matrix is typically about O(n^2.4) to O(n^3) (depending on the implementation)
- WARNING: the normal equation gets very slow when the number of features grows very large
- linear with the regards to the number of instances in the training set O(m)

Gradient Descent

- idea of gradient descent is to tweak parameters iteratively to minimize a cost function
- fill O (parameter vector) with random values and then make minor changes at each step to decrease the cost function (MSE) until the algorithm converges to a minimum
- learning rate hyperparameter determines the size of the steps
- small = long time and many iterations
- large - may overshoot the target
- not all cost functions are regular and smooth
  - may converge to local minimum rather than global minimum
- the MSE cost function for a Linear Regression model happens to be a convex function
  - if you pick any two points on the curve, the line segment joining them never crosses the curve
  - implies that there are no local minima, just one global minimum
  - also a continuous function with a slope that never changes abruptly
  - Gradient Descent is guaranteed to approach arbitrarily close the global minimum
    - learning rate can't be too high
    - must run adequate number of iterations

- WARNING: when using Gradient Descent, ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge

Batch Gradient Descent

- compute the gradient of the cost function with regards to each model parameter θj to implement Gradient Descent
    - e.g. calculate how much the cost function will change if change θj just a small amount
    - e.g. partial derivative.

```
Partial derivatives of the cost function here
```

- can use another equation to compute all at once

```
Gradient vector of the cost function
```

- WARNING: this formula involves calculations over the full training set X, at each Gradient Descent step
    - reason for name Batch Gradient Descent
    - uses the whole batch of training data at every step
    - very slow on very large training sets
    - Gradient Descent scales well with the number of features
    - training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation
- once the gradient vector is set, reverse the direction of calculations by subtracting V0MSE (O)

```
Gradient descent step
```

- implementation

```
eta = 0.1 # learning rate
n_iterations = 1000
m = 100
theta = np.random.randn(2,1) # random initialization
for iteration in range(n_iterations):
    gradients = 2 / m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

- can use grid search to find a good learning rate
    - limit the number of iterations so that grid search can eliminate models that take too long to converge
- to set the number of iterations (and avoid a value too low or high)
    - set a very large number of iterations

    - interrupt the algorithm when the gradient vector becomes very small
        - e.g. when its norm becomes smaller than a tiny number ϵ (called the tolerance)
        - happens when Gradient Descent has (almost) reached the minimum
- the convergence rate
    - when the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), it can be shown that Batch Gradient Descent with a fixed learning rate has a convergence rate of O(1/iterations)
    - e.g. if you divide the tolerance ϵ by 10 (to have a more precise solution), then the algorithm will have to run about 10 times more iterations

Stochastic Gradient Descent

- picks a random instance in the training set at every step and computes the gradients based only on that single instance
    - makes the algorithm much faster since it has very little data to manipulate at every iteration
    - makes possible to train on huge data sets
- much less regular than Batch Gradient Descent
    - over time ends up very close to minimum
    - final parameter values are good but not optimal
- randomness helps escape from local optima but keeps algorithm from settling at the minimum
- simulated annealing
- learning schedule - learning rate at each iteration

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index + 1]
        yi = y[random_index:random_index + 1]
        gradients = 2 * xi.T.dot(xi.dot( theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

- using sklearn

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter = 50, penalty = None, eta0 = 0.1)
sgd_reg.fit(X, y.ravel())
```

Mini Batch Gradient Descent

- at each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called mini-batches
- advantage over Stochastic GD is a performance boost from hardware optimization of matrix operations especially when using GPUs
- the algorithm's progress in parameter space is less erratic than with SGD
- Mini-batch GD will end up walking around a bit closer to the minimum than SGD
  - may be harder for it to escape from local minima

```
Algorithm       Large m     Out-of-core support     Large n     Hyperparams     Sc
Normal Eqn      Fast        No                      Slow        0               No
Batch GD        Slow        No                      Fast        2               Ye
Stochastic GD   Fast        Yes                     Fast        ≥ 2             Ye
Mini-batch GD   Fast        Yes                     Fast        ≥ 2             Ye
```

Polynomial Regression

- can use a linear model to fit nonlinear data
- simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree = 2, include_bias = False)
X_poly = poly_features.fit_transform(X)
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

- when there are multiple features, Polynomial Regression is capable of finding relationships between features (which is something a plain Linear Regression model cannot do)
  - PolynomialFeatures adds all combinations of features up to the given degree
  - e.g. if there were two features a and b, PolynomialFeatures with degree = 3 would not only add the features a2, a3, b2, and b3, but also the combinations ab, a2b, and ab2
- WARNING: PolynomialFeatures(degree = d) transforms an array containing n features into an array containing (n + d)! / d!n! features, where n! is the factorial of n

Machine Learning Project Checklist

process summary

1. get an overview of project

2. retrieve data
3. discover and visualize the data to gain insights
4. prepare the data for ML algorithms
5. select a model and train it
6. fine-tune the system
7. present solution
8. launch, monitor, and maintain system

1. get an overview of project

1. define the objective in business terms
    - how will the solution be used?
    - what are the current solutions/workarounds (if any)?
    - how should this problem be framed (supervised/unsupervised, online/offline, etc.)?
    - how should performance be measured?
    - is the performance measure aligned with the business objective?
    - what would be the minimum performance needed to reach the business objective?
    - what are comparable problems?
    - can you reuse experience or tools?
    - is human expertise available?
    - how would you solve the problem manually?
2. list the assumptions made
3. verify assumptions if possible

2. retrieve data

- NOTE: automate as much as possible

1. list the data needed and how much is needed
2. find and document where you can get that data
3. check how much space it will take
4. check legal obligations, and get authorization if necessary
5. get access authorizations
6. create a workspace (with enough storage space)
7. get the data
8. convert the data to a format you can easily manipulate (without changing the data itself)
9. ensure sensitive information is deleted or protected (e.g., anonymized)
10. check the size and type of data (time series, sample, geographical, etc.)

11. sample a test set, put it aside

3. discover and visualize the data to gain insights

Note: try to get insights from a field expert for these steps. Create a copy of the data for exploration (sampling it down to a manageable size if necessary). Create a Jupyter notebook to keep a record of your data exploration. Study each attribute and its characteristics: Name Type (categorical, int/ float, bounded/ unbounded, text, structured, etc.) % of missing values Noisiness and type of noise (stochastic, outliers, rounding errors, etc.) Possibly useful for the task? Type of distribution (Gaussian, uniform, logarithmic, etc.) For supervised learning tasks, identify the target attribute(s). Visualize the data. Study the correlations between attributes. Study how you would solve the problem manually. Identify the promising transformations you may want to apply. Identify extra data that would be useful (go back to “Get the Data”). Document what you have learned.

4. prepare the data for ML algorithms

Work on copies of the data (keep the original dataset intact). Write functions for all data transformations you apply, for five reasons: So you can easily prepare the data the next time you get a fresh dataset So you can apply these transformations in future projects To clean and prepare the test set To clean and prepare new data instances once your solution is live To make it easy to treat your preparation choices as hyperparameters Data cleaning: Fix or remove outliers (optional). Fill in missing values (e.g., with zero, mean, median…) or drop their rows (or columns). Feature selection (optional): Drop the attributes that provide no useful information for the task. Feature engineering, where appropriate: Discretize continuous features. Decompose features (e.g., categorical, date/ time, etc.). Add promising transformations of features (e.g., log(x), sqrt(x), x ^ 2, etc.). Aggregate features into promising new features. Feature scaling: standardize or normalize features.

5. select a model and train it

If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or Random Forests). Once again, try to automate these steps as much as possible. Train many quick and dirty models from different categories (e.g., linear, naive Bayes, SVM, Random Forests, neural net, etc.) using standard parameters. Measure and compare their performance. For each model, use N-fold cross-validation and compute the mean and standard deviation of the performance measure on the N folds. Analyze the most significant variables for each algorithm. Analyze the types of errors the models make. What data would a human have used to avoid these errors? Have a quick round of feature selection and engineering. Have one or two more quick iterations of the five previous steps. Short-list the top three to five most promising models, preferring models that make different types of errors.

6. fine-tune the system

You will want to use as much data as possible for this step, especially as you move toward the end of fine-tuning. As always automate what you can. Fine-tune the hyperparameters using cross-validation. Treat your data transformation choices as hyperparameters, especially when you are not sure about them (e.g.,   should I replace missing values with zero or with the median value? Or just drop the rows?). Unless there are very few hyperparameter values to explore, prefer random search over grid search. If training is very long, you may prefer a Bayesian optimization approach (e.g., using Gaussian process priors, as described by Jasper Snoek, Hugo Larochelle, and Ryan Adams). 1 Try Ensemble methods. Combining your best models will often perform better than running them individually. Once you are confident about your final model, measure its performance on the test set to estimate the generalization error. Warning Don't tweak your model after measuring the generalization error: you would just start overfitting the test

7. present solution

Document what you have done. Create a nice presentation. Make sure you highlight the big picture first. Explain why your solution achieves the business objective. Don't forget to present interesting points you noticed along the way. Describe what worked and what did not. List your assumptions and your system's limitations. Ensure your key findings are communicated through beautiful visualizations or easy-to-remember statements (e.g., "the median income is the number-one predictor of housing prices").

8. launch, monitor, and maintain system

Get your solution ready for production (plug into production data inputs, write unit tests, etc.). Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops. Beware of slow degradation too: models tend to "rot" as data evolves. Measuring performance may require a human pipeline (e.g., via a crowdsourcing service). Also monitor your inputs' quality (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems. Retrain your models on a regular basis on fresh data (automate as much as possible).