CNT 6154

Advanced Applied Machine Learning Manual

Book Reference: Hands-On Machine Learning with Scikit Learn & Tensorflow (O'Reilly) by *Aurélien Géron*

Table of Contents

Chapter 1: The Machine Learning Landscape

What is Machine Learning?

- Machine Learning (ML) is a field of study that gives computers the ability to learn from the data without being explicitly programmed.
- "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."
- Ex: Marking emails as spam and non-spam (also known as "ham"). ML algorithms can assist in flagging the email by using examples given by the users to label them as spam or ham. The examples that the system uses to learn is called a labeled *training set* and training examples are known as a *training instance*, or *sample*.

Why do we use Machine Learning?

- Currently, certain problems require the manual coding of a long list of rules. This is very time consuming. A Machine Learning algorithm can instead learn from the data and save time and effort
- Certain problems do not have a solution while taking a traditional approach. ML algorithms can potentially give a solution.
- In fluctuating environments, a Machine Learning system can adapt and learn from new data.
- ML models can help us gather insights from large and complex datasets.

Types of Machine Learning Systems:

- Supervised, Unsupervised, Semisupervised, and Reinforcement Learning – training with human supervision or not
- Online vs batch learning - learning incrementally or on the spot
- Instance-based and Modeled-based learning - comparing new data to the known data or instead detecting patterns in the training data and predicting the model

A. Supervised/Unsupervised Learning

- In *supervised learning*, you provide the algorithm with a set of training data and the solution, or the result, you are looking for. This solution, or the result, is known as *label*.
- *Classification* and *Regression* are common examples of supervised learning.
- In *unsupervised learning*, the training data is unlabeled, which means the system tries to learn on its own. *Clustering, visualization, dimensionality reduction*, and *association rule learning* are types of unsupervised learning.
- *Semisupervised learning* algorithms can deal with partially labeled data containing a lot of unlabeled data and little labeled data.
- *Reinforcement learning* is when an agent selects and performs actions. It gets rewards in return. The agent must then learn the best strategy (called *policy*) to get the most reward over time. A policy defines the best action in a given situation.

Some of the supervised learning algorithms are:

1) K-Nearest Neighbors
2) Linear Regression
3) Logistic Regression
4) Support Vector Machine (SVMs)
5) Decision Trees and Random Forests
6) Neural Networks

B. Batch and Online Learning

- In *batch learning*, you train the algorithm with all the available data at once. The system is trained and launched into production immediately and does not learn anymore, it simply applies what was learned based on the available data. This is known as *offline learning.* Offline learning is good for data which is not updating hourly or daily.
- An out-of-core learning algorithm chunks the data into mini-batches and utilizes online learning techniques to learn from these mini-batches.
- *Online learning* feeds the data incrementally in small groups called *batches*. This helps save memory and time, since the system can learn about the new data as it arrives.
- Online learning algorithms load the data, then run the training steps on it and repeat the process on all the data sets. Each learning step is quick and inexpensive. The system can learn on the fly.

C. Instance-Based vs. Model-Based Learning

- Another way to categorize Machine Learning systems is by how they *generalize.* This means that given a set of training examples, the system should be able to generalize the examples it has never seen before. The goal is to perform well on new instances of data.
- *Instance-Based learning*: the system learns the example by heart, and then generalizes the algorithm to new cases using similar measures.
- *Model-Based learning:* takes the set of examples and builds a model of these examples to make predictions. First select a model, (ex: a linear model) and then specify performance measure by defining a cost function or fitness function, which measures how good or bad your model is.

Summary of how Machine Learning works:

1) Study the data
2) Select a model
3) Train the model on the training data
4) Apply the model to make predictions on new cases

Main Challenges of Machine Learning:

- **Insufficient data**: Machine Learning algorithms need a lot of data to give desirable results. Even simple problems need thousands of training samples. For complex problems, such as image or speech recognition, you may need millions of training samples.
- **Non-representative training data**: Training data must be representative of the new cases we want to generalize to. This is not easy - if the sample is too small, there may be sampling noise

and even very large samples can be non-representative if the sampling method is flawed. This is called sampling bias.

- **Poor Quality data**: If your training dataset has erroneous entries, outliers, or noise, the system may not detect the underlying patterns accurately. Hence, cleaning the training data is very important. Data scientists spend a significant part of their time cleaning the data
- **Irrelevant features**: The system will only be capable of learning if the training data contains enough relevant features. A good set of training features is very important for a successful model. The process creating relevant features is called feature engineering.
- **Overfitting the training data**: If model performs very well on the training data, but does not generalize to out of sample data, it is called overfitting. Advanced models, such as deep neural networks, can learn subtle patterns in the training data, but if the data is noisy, or if the number of samples is too small, then the model is likely to learn the noise itself. Possible solutions to this is getting more data, simplifying the model, or reducing the noise in the training data.
- **Underfitting the data**: It is the opposite of overfitting, where the model is too simple and does not satisfactorily learn the underlying structure of the data. In such a case, the predictions are inaccurate, even on the training examples.

Testing & Validating:

Model Generalization:

To find out how well a model generalizes, it needs to be tested on out of sample data. The training data is split into 2 sets: the *training set* and *test set*. The best way to do this is by splitting your data into two sets: *training set* and *test set*. Generally, you split 80% of data for the training set and 20% of data for the test set. By evaluating your model based on the data, you will come to know how well your model will perform. The error rate (generalization error or test error), is an indicator of how well the model generalizes.

- If the training error is low, that means your model is performing well. If the error rate is high, that means your model is *overfitting* the training data.
- If we optimize the model to give a very low test error, the model has inadvertently been trained on the test data as well, and may not generalize well on data outside the train and test sets.
- The opposite case is when your data is *underfitted*, which occurs when your model is too simple to learn the underlying structure of the data.
- To tackle this, a solution is to split the training data into 3 sets: the training set, validation set and the test set. The hyperparameter of the model is fine-tuned to minimize the error on the validation set. A hyperparameter is a parameter of the learning algorithm itself, not of the model. The test set is not used until the model is completely optimized. The final run on the test data will be a good indicator of the generalization error.
- Cross-validation is a technique that helps compare models (for model selection and hyperparameter tuning) without the need for a separate validation set, which helps save data.

Ways to fix the problem:

1) Select a more powerful model, with more parameters
2) Provide better features to the learning algorithm
3) Minimize constraints on the model

There is no model that is guaranteed to work well on any given dataset. The only way to find out is to evaluate all the models. However, it is not computationally possible to test all the models. Hence, a reasonable assumption is made about the data and only relevant models are evaluated. For example, for simple problems with simple relationships, linear models with various levels of regularization can be tested. For more complex relationships, we can evaluate neural networks.

Chapter 2: End-to-End Machine Learning Project

Steps to work through as a data scientist:

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system

Project Steps:

1) Look at the big picture

- What is the business objective and how does the organization expect to use this model? This will determine how we frame the problem, what algorithms to test, what measures we use to evaluate your model, and how to optimize.
- Is there an existing solution and if so, what does it look like? It gives the data scientist a reference or baseline. It may also provide some insights on solving the problem.
- *Frame the problem:* Supervised or unsupervised? Classification or regression? Batch learning or online learning?
- Select a performance measure. Several regression problems use RMSE (root mean square error). It gives an indication of how much error the system has in its predictions. It holds a higher weight for large errors.
- List and verify all the assumptions about the problem and the data at hand.

2) Get the data

- Create the workspace:
  - Install python
  - Create a workspace directory
  - Install Jupyter, NumPy, Pandas, Matplotlib and Scikit-Learn (either by installing Anaconda or using pip)
- Download the data. Data may be available in any of the sources mentioned below. Sometimes pandas functions such as read_csv() may do the job. Sometimes we may need to use libraries such as pyodbc. In most cases we store the data in a pandas dataframe.
  - A relational database
  - Spread across multiple tables or files
  - Available on a remote server
  - May need to be scraped from the web
- Take a quick look at the data structure:
  - Look at the top 5 rows of a dataframe (using the .head() method)
  - Look at a quick description of data (using the .info() method)

3) Discover and visualize the data to gain insights

- Do a quick check of the statistical summary of the variables (using the .describe() method).
- Check the histograms of all the variables (using the .hist() method in matplotlib).
- Create a correlation matrix to check the correlations in a quantitative way (using the .corr() method in pandas).
- A heatmap is another way to visualize the correlations.
- Do a scatter matrix plot to visualize the relationships between the independent variables and also their correlation with the dependent variable. Sometimes the correlation can be zero, but the visualization can suggest non-linear relationships.
- If you have geographical data, visualize it on a map.

4) Data Cleaning - Prepare the data for Machine Learning algorithms

- *Dealing with null values:* Depending on the problem at hand, either drop the null values or impute them. There are several ways to impute missing values
    - Impute with mean
    - Impute with median
    - Use the non-null values to predict the null values
- Categorical and text variables need to be converted to numerical values to be used for training ML models
    - For ordered categories, ordinal encoding can be a good fit. It considers that the numerical values closer to each other are more similar
    - For categories which are not ordered, one-hot encoding is well suited. It is also one of the most popular encoding schemes
- Feature scaling; ML algorithms may not give an optimal performance when the variables are on very different scales.
  2 common ways to rescale the variables are:
    - *Min-max scaling:* The values are rescaled to be in the range of 0 to 1. The min value is mapped to 0 and the max value is mapped to 1.
    - *Standardization:* The values are rescaled to have a mean of 0 and a standard deviation of 1.

5) Prepare the data

- Split the dataset into train and test sets – so that the model learns from the training data and we get to check its effectiveness on the test set. The samples in the test set are not used for training.
- It is typical to use 80% of the data from training and 20% for test. But there is no hard and fast rule. The percentage of train-test split is selected based on the dataset at hand.
- Scikit-learn provides a function train_test_split() to automatically split the dataset in the specified ratio. This function uses a purely random way of splitting the data.
- If all the classes in the dataset do not have the same number of samples, then a method called *stratified sampling* is used to avoid sampling bias.
- In whatever percentage we split the dataset, some samples are not used for training. To overcome this, a method called *cross-validation* is used.
    - *K-fold cross-validation:* randomly splits the training set into k distinct subsets called *folds*, then it trains and evaluates the Decision Tree model k times, picking

a different fold for evaluation every time and training on the other k-1 folds. The result is an array containing the k evaluation scores.

- *Cross-validation* allows us to get an estimate of the performance of the model. It also gives a measure of how precise the estimate is, in terms of the standard deviation.
- A sequence of data processing steps is called a *data pipeline*. Pipelines are very common in ML systems, since there are a lot of data manipulation steps.

6) Select a model and train it

- Select features and train a model that seems appropriate (linear regression of the underlying distribution is Gaussian and the relationship seems linear)
- Make an estimate of the error. Typically estimates such as RMSE (Root Mean Square Error) or MAE (Mean Absolute Error) are used.
- If the error seems too high, then it can mean that the model not sufficiently explain the variance in the dependent variable. In such a scenario, we try to do one or more of the following:
    - Train a more powerful model since the earlier model is too simple
    - Add more features since the initially selected features may be insufficient
    - Reduce the constraints on the model if any (such as regularization)
- If the error is almost zero, then it may mean that the model is overfitting. Possible solutions:
    - Simplify the model
    - Add constraints on the model to reduce overfitting (use techniques like regularization)
    - Increase the amount of training data
- Try to fit several models without spending too much time tuning the hyper-parameters and short list the promising ones. Save the models and hyper-parameter values for later use.

7) Fine-tune your model

- The hyper-parameters of the promising models need to be optimized to obtain the best accuracy.
- The fine tuning can be done manually to find the right combination of hyper-parameters or can be done using a function such as `GridSearchCV`.
    - A matrix of possible values is input to the `GridSearchCV` function and it evaluates all the possible combinations using cross-validation. The combination with the lowest RMSE (or the chosen error estimate) is the most optimal combination.
- The grid search option does not scale well when testing a large number of combinations and can take a really long time to compute, while having a limitation of trying only a few values for each hyper-parameter.
    - A function such as `RandomSearchCV` evaluates a specified number of random combinations of randomly selected values of hyper-parameters.
    - This gives the user more control on the duration while testing more combinations. For example run 1000 iterations of random combinations and choose the best one instead of trying few fixed values.
- Ensemble methods: Combining the most promising models to obtain a better performance than the individual models is called an Ensemble method.

- Example is random forests which is a combination of individual decision trees

8) Evaluate the model on the test set

- After fine tuning, we have a model that can be evaluated on the test set.
- The performance on the test set is slightly worse than the training or validation sets.
- However, if the performance on the test set is much worse than training then it indicates an overfit model.
- Prelaunch: Present your solution to highlight what we learned, what worked, the assumptions and the limitations. It is good practice to document everything and create good visualizations.

9) Launch, monitor, and maintain your system

- Get your solution ready for production. Use the production data for training and write unit tests.
- Write monitoring code to check the performance. It should trigger alerts when the performance drops.
- Sample the systems predictions and evaluate them for accuracy.
- Evaluate the quality of the training data. Performance may degrade due to poor quality of input data.
- Training the models on a regular basis using new data to keep the model updated. It is best to automate this process.

Chapter 3: Classification

MNIST

- The MNIST dataset is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents.
- It is a classic example for machine learning practice for classification, to identify each image as a specific number.

Training a Binary Classifier

- A *binary classifier* will be able to distinguish between just two classes. In the MNIST dataset, we can simply by trying to classify is a number is "5" or not.
- First, create the target vectors for this classification.
- Next select a classifier.
    - *Stochastic Gradient Descent* (SGD) classifier, using Scikit-Learn's SGDClassifier class. This classifier has the advantage of being capable of handling very large datasets efficiently.
    - SGD deals with training instances independently, one at a time (which also makes SGD well suited for *online learning*
- When predicting the number 5 from the image, the output will read as True or False.

Performance Measures

Measuring Accuracy Using Cross-Validation

- K-fold cross-validation means splitting the training set into K-folds, then making predictions and evaluating them on each fold using a model trained on the remaining folds.
- Accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with *skewed datasets*.

Confusion Matrix

- A better way to evaluate the performance of a classifier is to look at the *confusion matrix.*
- Count the number of times instances of class A are classified as class B.
    - *MNIST example:* to know the number of times the classifier confused images of 5s with 3s, you would look in the 5th row and 3rd column of the confusion matrix.
- To compute the confusion matrix, you first need to have a set of predictions, so they can be compared to the actual targets.
- `cross_val_predict()` performs K-fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold.
- Each row in a confusion matrix represents an *actual class*, while each column represents a *predicted class*.
- A perfect classifier would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal (top left to bottom right).

- Precision is typically used along with another metric named *recall*, also called *sensitivity* or *true positive rate* (*TPR*): this is the ratio of positive instances that are correctly detected by the classifier.

Precision and Recall

- It is convenient to combine precision and recall into a single metric called the *F1 score*, in particular if you need a simple way to compare two classifiers.
- The F1 score is the *harmonic mean* of precision and recall.
- The regular mean treats all values equally, while the harmonic mean gives much more weight to low values.
- Therefore, the classifier will only get a high F1 score if both recall and precision are high.
- The F1 score favors classifiers that have similar precision and recall.
- Increasing precision reduces recall, and vice versa. This is called the *precision/recall tradeoff*.

Precision/Recall Tradeoff

- For each instance, the SGDClassifier computes a score based on a *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.
- *Determining which threshold to use:* You first need to get the scores of all instances in the training set using the cross_val_predict() function again, but this time specifying that you want it to return decision scores instead of predictions.
- With these scores you can compute precision and recall for all possible thresholds using the precision_recall_curve() function.
- You can plot precision and recall as functions of the threshold value using Matplotlib, and then select he threshold value that gives you the best precision/recall.
- Another way to select a good precision/recall tradeoff is to plot precision directly against recall.

The ROC Curve

- *Receiver operating characteristic* (ROC) curve is another common tool used with binary classifiers.
- It is similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate.*
- FPR: the ratio of negative instances that are incorrectly classified as positive. It is equal to one minus the *true negative rate* (TNR, which is also called *specificity*), which is the ratio of negative instances that are correctly classified as negative.
    - ROC curve plots *sensitivity* (recall) versus 1 – *specificity*.
- To plot the ROC curve, you first need to compute the TPR and FPR for various threshold values, using the roc_curve() function.
- Then plot the FPR against the TPR using Matplotlib.
- The higher the recall (TPR), the more false positives (FPR) the classifier produces.
- One way to compare classifiers is to measure the *area under the curve* (AUC). A perfect classifier will have a *ROC AUC* equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5

Multiclass Classification:

- Binary classifiers distinguish between two classes, while *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.
- Random Forest classifiers or naive Bayes classifiers can handle multinomial classifiers, while Support Vector Machine classifiers or Linear classifiers are only binary classifiers.
- However, you can perform multiclass classification using multiple binary classifiers.
- *One-vs-All (OvA) / One-vs-The-Rest Strategy:*
    - To create a system that can classify the digit images into 10 classes (from 0 to 9), then train 10 binary classifiers, one for each digit (a 0-detector, 1-detector, a 2-detector, etc.)
    - To classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score
- One-vs-One (OvO) strategy:
    - Train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, etc.
    - For N classes, you need to train $N \times (N - 1) / 2$ classifiers
    - OvO Advantage: each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish.
- Scikit-Learn detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvA.
- Make sure to evaluate these classifiers by using cross-validation.

Error Analysis:

- Once you find a promising model, you will want to find ways to improve it. One way to do this is to analyze the types of errors it makes.
- First, you can look at the confusion matrix.
- Then, you need to divide each value in the confusion matrix by the number of images in the corresponding class, so you can compare error rates instead of absolute number of errors (which would make abundant classes look unfairly bad).
- Afterward, fill the diagonal with zeros to keep only the errors, and plot the result.
- Analyzing the confusion matrix can often give you insights on ways to improve your classifier.
- Analyzing individual errors can also be a good way to gain insights on what your classifier is doing and why it is failing, but it is more difficult and time-consuming.

Multi-label Classification:

- In some cases you may want your classifier to output multiple classes for each instance.
- A classification system that outputs multiple binary labels is called a *multilabel classification* system.
- There are many ways to evaluate a multilabel classifier, and selecting the right metric really 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.
    - This assumes that all labels are equally important, which may not be the case.
    - One option is to to give each label a weight equal to its *support* (i.e., the number of instances with that target label).

Multi-output Classification:

- *Multioutput-multiclass classification* (or simply *multioutput classification*): a generalization of multi-label classification where each label can be multiclass (i.e., it can have more than two possible values).
- Example of a multioutput classification system:
    - Build a system that removes noise from images.
    - It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255).

Chapter 4: Training Models

Linear Regression:

- A linear model makes a prediction by computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*).
- The most common performance measure of a regression model is the Root Mean Square Error (RMSE). Therefore, to train a Linear Regression model, you need to find the value of θ that minimizes the RMSE.
- It is simpler to minimize the Mean Square Error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root).

Linear Regression model prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Linear Regression model prediction (vectorized form)

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

MSE cost function for a Linear Regression model

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

The Normal Equation:

- Gradient Descent (GD) has been used for linear regression to minimize the cost function of θ, by running multiple iterations of the gradient descent to converge to the global minimum.
- With the normal equation, we can solve for θ analytically, so rather than needing to run an iterative algorithm, we can instead solve for the optimal value of θ all in one go, so that in one step you can get the optimal value.
- Normal equation: a *closed-form solution* (a mathematical equation that directly gives the result) that finds the value of θ that minimizes the cost function.

Normal Equation

$$\hat{\boldsymbol{\theta}} = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{y}$$

- $\hat{\boldsymbol{\theta}}$ is the value of θ that minimizes the cost function.
- y is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

- To compute for the value of θ, use the inv() function from NumPy's Linear Algebra module (np.linalg) to compute the inverse of a matrix, and the dot() method for matrix multiplication.
- Once you have results for θ, you can than make predictions using θ, and next perform linear regression using Scikit-Learn.
- You can calculate the pseudoinverse with a standard matrix factorization technique called *Singular Value Decomposition* (SVD) which decomposes the training set matrix X into the matrix multiplication of three matrices.
- The pseudoinverse is more efficient than computing the normal equation, since the normal equation may not work in some cases, but the pseudoinverse is always defined.

Computational Complexity:

- Normal Equation computes the inverse of $X_T X$, which is an $(n + 1) \times (n + 1)$ matrix (n is the number of features)
- The *computational complexity* of inverting this matrix is about $O(n^{2.4})$ to $O(n^3)$, which means double the number of features, and it will multiply the computation time by roughly $2^{2.4}$.
- The SVD approach used by Scikit-Learn's LinearRegression class is about $O(n^2)$, which doubles the computation time by 4.
- When using a trained Linear Regression model, the computational complexity is linear, meaning making predictions on twice as many instances (or twice as many features) will just take roughly twice as much time.

<u>Gradient Descent:</u>

- A very generic optimization algorithm capable of finding optimal solutions to a wide range of problems, tweaking parameters iteratively in order to minimize a cost function.
- Start the algorithm by filling θ with random values (called *random initialization*), and then improve it gradually, taking one step at a time, each step attempting to decrease the cost function (i.e. the MSE), until the algorithm *converges* to a minimum.

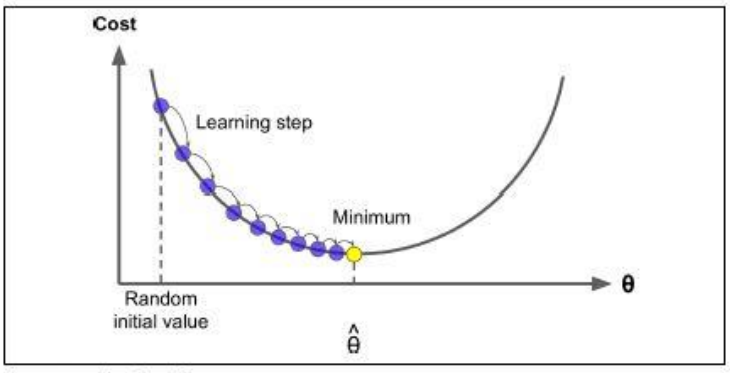

Figure 4-3. Gradient Descent

- As seen above, the size of the steps, determined by the *learning rate* hyperparameter, is important for the Gradient Descent. As a refresher, a hyperparameter is a parameter of a learning algorithm (not of the model). As a result, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training.

- If the learning rate is too small, then the algorithm has to go through many iterations to converge, taking a long time.

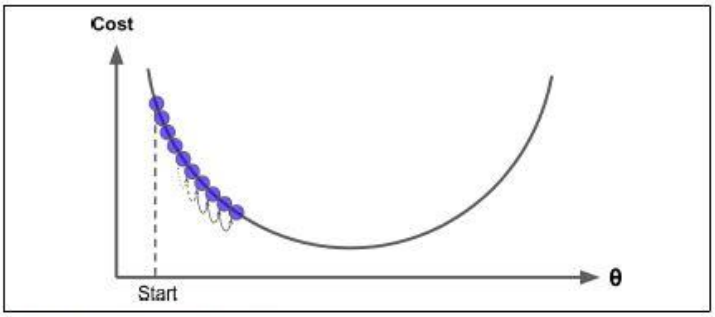

Figure 4-4. Learning rate too small

- If the learning rate is too large, the algorithm will diverge and fail to find a good solution.

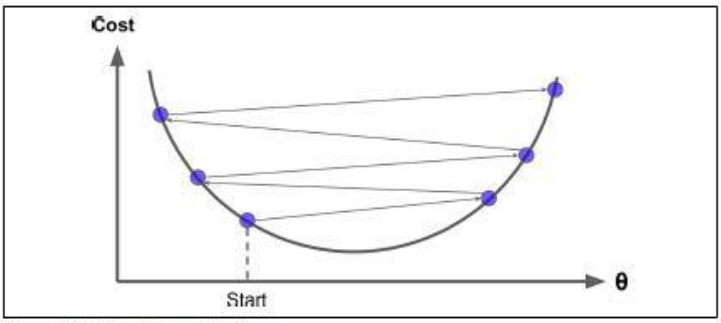

Figure 4-5. Learning rate too large

- Two challenges of Gradient Descent:
  - If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*.
  - If it starts on the right, then it will take a very long time to cross the plateau, and if you stop too early, you will never reach the global minimum.
  - The MSE cost function for a Linear Regression model happens to be a *convex function*, so for any two points on the curve, the line segment joining them never crosses the curve. Therefore there are no local minima, just one global minimum.

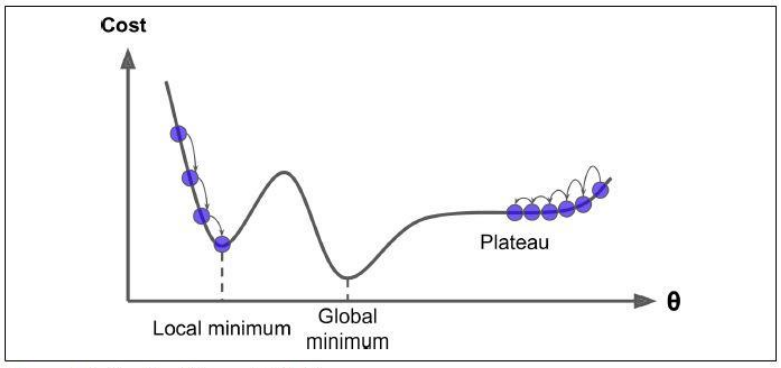

Figure 4-6. Gradient Descent pitfalls

Batch Gradient Descent:

- For Batch Gradient Descent, compute the gradient of the cost function in relation to each model parameter θ_i. Then you need to calculate how much the cost function will change if you change θ_i just a little bit. This is called a *partial derivative*.
- Batch gradient descent computes the gradient using the whole dataset.
    - This is great for convex or relatively smooth error manifolds, which help with the debugging of machine learning models.
    - In this case, we move somewhat directly towards an optimum solution, either local or global.
    - Additionally, batch gradient descent, given an annealed learning rate (which is different ways of decreasing the step size), will eventually find the minimum located in its basin of attraction.
        - One popular way to decrease learning rates is by steps: to simply use one learning rate for the first few iterations, then drop to another learning rate for the next few iterations, and then drop the learning rate further for the next few iterations.
        - Another way would be to linearly decrease the learning rate with each iteration.

Stochastic Gradient Descent (SGD):

- In SGD, we are using the cost gradient of **1 example** at each iteration, instead of using the sum of the cost gradient of ***all*** examples.
- In SGD, before for-looping, you need to randomly shuffle the training examples.
- In SGD, because it's using only one example at a time, its path to the minima is noisier (more random) than that of the batch gradient. However that's okay as we are indifferent to the path, as long as it gives us the minimum *and* the shorter training time.

Mini-batch Gradient Descent:

- Mini-batch gradient descent uses **n** data points (instead of **1** sample in SGD) at each iteration.
- Most applications of SGD actually use a minibatch of several samples. Single samples are really noisy, while minibatches tend to average a little of the noise out.
- Ideally the minibatch size should be small enough to avoid some of the poor local minima, but large enough that it doesn't avoid the global minima or better-performing local minima.

Polynomial Regression:

- *Polynomial regression* uses a linear model to fit non-linear data by adding the powers of each feature as new features, then train a linear model on this extended set of features.

Original equation: $y = 0.5x_1^2 + 1.0x_1 + 2.0$

After polynomial regression, the model predicts: $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$

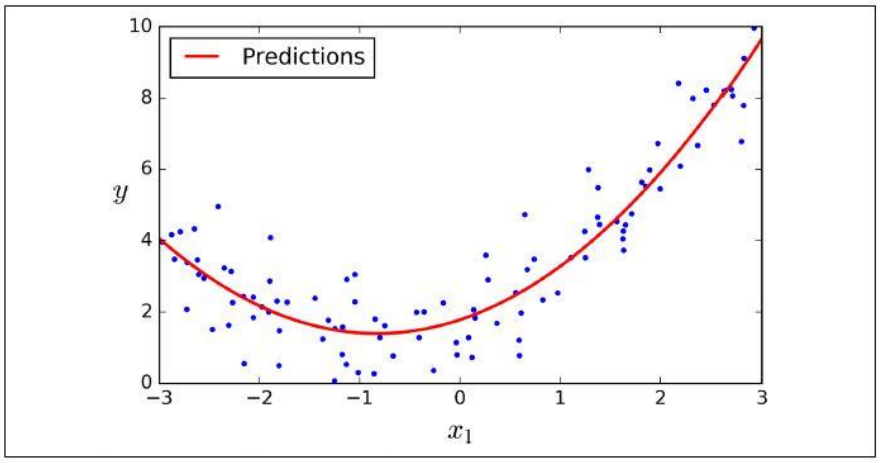

Figure 4-13. Polynomial Regression model predictions

Learning Curves:

- With high-degree polynomial regression, you can fit the training data much better than with regular linear regression.
- However, with high-degree polynomial regression you'll likely be overfitting the training data, while with linear regression, it'll be underfitting.
- *Learning Curve:* the plots of a model's performance on the training set and the validation set as a function of the training set size (or training iteration).
- To make the plots, train the model several times on different subsets of the training set.

Regularized Linear Models:

- Regularization is useful because it helps reduce overfitting; the fewer degrees of freedom a model has, the harder it will be for it to overfit the data.
- With linear models, regularization is usually done by constraining the weights of the model.
- A model with some regularization typically performs better than a model without any regularization

Ridge Regression:

- *Ridge Regression* (also called *Tikhonov regularization*) is a regularized version of Linear Regression: a *regularization* term equal to $\alpha\sum_{i=1}^{n}\theta_i^2$ is added to the cost function.
- This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible.
- The regularization term should only be added to the cost function during training.
- After the model is trained, evaluate the model's performance with the unregularized performance measure.
- Similar to Linear Regression, Ridge Regression can be done by computing a closed-form equation or by performing GD.

Lasso Regression:

- *Lasso Regression* (also known as Least Absolute Shrinkage and Selection Operator Regression) is another regularized version of Linear Regression. Similar to Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ1 norm of the weight vector instead of half the square of the ℓ2 norm.
- Lasso Regression tends to completely eliminate the weights of the least important features, such as setting them to zero. It automatically performs feature selection and outputs a *sparse model*, with few non-zero feature weights.

Elastic Net:

- Elastic Net in the middle between Ridge Regression and Lasso Regression. The regularization term is a mix of both Ridge and Lasso's regularization terms, and the mix ratio *r* is controlled.
    - When *r* = 0, Elastic Net is equivalent to Ridge Regression.
    - When *r* = 1, it is equivalent to Lasso Regression.

Early Stopping:

- Early stopping is when you stop training on regularizing iterative learning algorithms (i.e. GD) as soon as the validation error reaches a minimum.
- If the validation error is higher than the training error that likely means you're overfitting the model. Fix by reducing the polynomial degree, increasing the training set, or regularizing the model.

Logistic Regression:

- *Logistic Regression*: estimate the probability that an instance belongs to a particular class.

Equation 4-14. Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

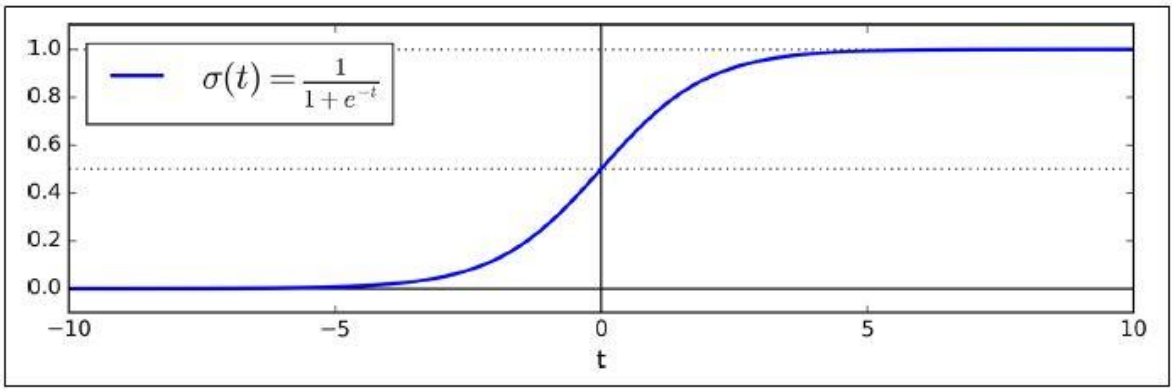

Figure 4-21. Logistic function

- As seen in the graph above, if the estimated probability if greater than 50%, then the model predicts that the instance belongs to the positive class "1" or it's less, then it belongs to negative class "0" as a binary classifier.

Estimating Probabilities:

- Logistic Regression model computes a weighted sum of input features (plus a bias term) but outputs the *logistic* (a sigmoid function that outputs a number between 0 and 1) of the result instead of the result directly like the Linear model

Training and Cost Function:

- Training aims to set the parameter vector θ so the model estimates high probabilities for positive instances (y = 1) and low probabilities for negative instances (y = 0).
- *Since – log(t) will be very large when t gets closer to 0, the cost function will be large if the model estimates a probability close to 0 for a positive instance and it will be very large if the model estimates a probability close to 1 for a negative instance.*
    - The cost function *maps event or values of one or more variables onto a real number.*
    - *The cost function over the whole training set is the average cost over all training instances.*
- There is nothing equal to the Normal Equation that is like a closed-form equation to compute θ to minimize the cost function, however, the cost function is convex, so GD (or other optimization algorithms) can find the global minimum, so long as the learning rate is not too big.

Decision Boundaries:

- A *decision boundary* is the threshold value or tipping point which we will classify values into a particular class if it is above or below that.
- Once the dataset is trained, the logistic regression classifier can estimate the probability that an item is a particular label.
- Logistic regression models can be regularized using l_1 or l_2 penalties.
    - A regression model that uses L1 regularization technique is called *Lasso Regression* and model which uses L2 is called *Ridge Regression.*
    - **Ridge regression** adds "*squared magnitude*" of coefficient as penalty term to the loss function.
    - **Lasso Regression** (Least Absolute Shrinkage and Selection Operator) adds "*absolute value of magnitude*" of the coefficient as a penalty term to the loss function.

Softmax Regression:

- *Softmax Regression* (Multinomial Logistic Regression): generalization of logistic regression that we can use for multi-class classification (under the assumption that the classes are mutually exclusive).
- In contrast, we use the (standard) Logistic Regression model in binary classification tasks.
- With instance x, the softmax regression model first computes a score $s_k(x)$ for each class k, then estimates the probability of each class with the softmax function (normalized exponential) to the scores.
- Softmax regression classifier, like the logistic regression classifier, predicts the class with the highest estimated probability (class with the highest score).

Chapter 5: Support Vector Machines

- Support Vector Machines (SVM) are one of the most popular, powerful, and versatile models of machine learning (ML), they can perform linear or nonlinear classification, regression, and outlier regression.

Linear SVM Classification:

- Linear SVM is a very fast ML algorithm for solving multiclass classification problems from ultra large data sets.
- An SVM classifier is like fitting the widest possible "street" between different classes, known as *large margin classification*. The key is to have the largest possible margin between the decision boundary which is determined by the instances located on the edges.

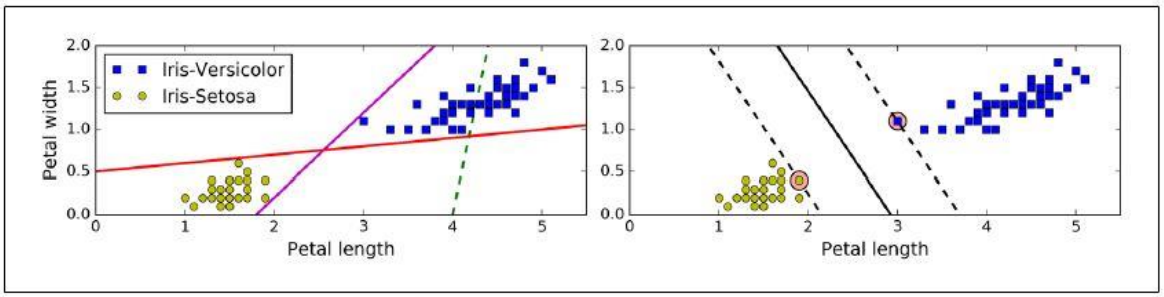

Figure 5-1. Large margin classification

In the graph above, the "widest street" is shown in the right graph, represented between the two dashed lines.

- If the training set is not scaled, the SVM will typically neglect small features.
- Adding more training instances "off the street" will not affect the decision boundary, it is full determined by the instances (known as *support vectors*) located on the edges.
- Any instance that is not a support vector, being off the street, will have no influence on the decision boundary.

Soft Margin Classification:

- *Hard Margin Classification:* when it's imposed for all the instances to be off the street and to the right hand side
- Hard margin SVM can work only when data is completely linearly separable without any errors (noise or outliers)
- Two issues with hard margin classification:
    - Only works when the data is completely linearly separable
    - It is sensitive to outliers
- It's better to use a more flexible model, while limiting *margin violations* (instances in the middle of the boundary or ones on the wrong side).

- Despite not converging as fast as the LinearSVC class, one can use SGDClassifier (which applies SGD to train a linear SVM classifier) when handling large datasets that might not fit in memory (out-of-core training), or handling online classification tasks.

Nonlinear SVM Classification:

- Linear SVM classifiers are efficient but many datasets are not linearly separable.
- A way to handle non-linear data, is to add more features (such as polynomial features), and sometimes this can result in a more linear result.

Polynomial Kernel:

- Adding polynomial features is an easy to implement and can work well with many types of ML algorithms, not just SVMs. However at a low polynomial degree, it cannot deal well with very complex datasets, and with a high polynomial degree, it creates a large number of features, which slows down the model.
- With SVMs, you can apply the *kernel trick*, which allows you to add many polynomial features (even with high degree polynomials), without having to add them.
    - The kernel trick is implement by the SVC class, as in the code below:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

    - The code above trains an SVM classifier using a 3rd degree polynomial kernel. If the model is overfitting, you can reduce the polynomial degree, and vice versa if it is underfitting. The hyperparameter coef0 controls how much the model is controlled by high degree polynomials vs. low degree polynomials.

Adding Similarity Features:

- Another way to handle non-linear problems is to add features computed using a *similarity function*, which measures how much each instance resembles a particular *landmark*.
- To select a landmark, create one at the location of each instance in the dataset. This will create many dimensions and thereby increases the chances that the transformed training set will be linearly separable.
    - The pitfall of this however, is that a training set with *m* instances and *n* features gets transformed into a training set with *m* instances and *m* features (if you drop the original features). So if the training set is very large, you will end up with many features.

Gaussian RBF Kernel:

- The Gaussian Radial Basis Function (RBF) kernel is in the form of a radial basis function (more specifically, a Gaussian function). The RBF kernel is defined as:

$$K_{\text{RBF}}(\mathbf{x}, \mathbf{x}') = \exp\left[-\gamma \|\mathbf{x} - \mathbf{x}'\|^2\right]$$

    - Parameter γ sets the "spread" of the kernel.
    - Remember that a kernel expresses a measure of similarity between vectors. The RBF kernel represents this similarity as a decaying function of the distance between the vectors (i.e. the squared-norm of their distance).
- The similarity features method is useful with any ML algorithm, but it may be computationally costly to compute all the additional features, for example on large training sets.
- The kernel trick makes it possible to obtain similar results as when many similarity features are added, without having to add them.
- For the Gaussian RBF (radial basis function) kernel, models are trained with different values of hyperparameters gamma (γ) and *C.*
    - Increase in gamma makes the instance's range of influence smaller – decision boundary becomes more irregular
    - Decrease in gamma makes the instance's range of influence larger – decision boundary becomes smoother.
- If an SVM classifier trained with an RBF kernel underfits the training set, it might be due to too much regularization. To fix this, increase gamma or C, or both.

Computational Complexity:

- The LinearSVC class implements an optimized algorithm for linear SVMs, but does not work with the kernel trick, its training complexity is approx. O(m × n) – number of training instances times number of training features.
- The algorithm is controlled by the tolerance hyperparameter ϵ (called "tol" in Scikit-Learn).
- The SVC class however, supports the kernel trick, the training time complexity is typically between $O(m_2 \times n)$ and $O(m_3 \times n)$
    - This means the algorithm gets very slow when the number of training instances increases; therefore the SVC class is better for small and medium sized data sets.
    - It scales well with the number of features, especially with sparse features, such as when each instance has few non-zero features.

SVM Regression:

- The SVM algorithm supports:
    - Linear & non-linear classification
    - Linear & non-linear regression
- SVM Classification: fit the largest possible street between two classes while limiting margin violations
- SVM Regression: fit as many instances as possible on the street while limiting margin violations (meaning, instances that are off the street); the street width is controlled by a hyperparameter
    - Adding more training instances within the margin does not affect the model's predictions; the model is ϵ-*insensitive.*

Under the Hood:

Decision Function and Predictions:

- The linear SVM classifier model predicts the class of a new instance x by computing the decision function $w_T x + b = w_1 x_1 + \cdots + w_n x_n + b$
    - If the result is positive, the predicted class is the positive class (1)
    - If the result is negative, the predicted class is the negative class (0)
- Training a linear SVM classifier will find the value of *w* and *b* that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

Training Objective:

- The smaller the weight vector *w*, the larger the margin, therefore in order to avoid any margin violation (hard margin), the decision function needs to be greater than 1 for all positive training instances, and lower than -1 for negative training instances.
- The hard margin linear SVM classifier objectice can be expressed as the *constrained optimization*.
- To obtain the soft margin objective, introduce a *slack variable* for each instance, it measures how much the i_{th} instance is allowed to violate the margin
- The two objectives are to make the slack variables as small as possible to reduce margin violations, and make ½ w_T w as small as possible to increase the margin.
    - Since these two objectives are in conflict with each other, the hyperparameter C can define the trade-off between these two objectives.

Quadratic Programming:

- The hard margin and soft margin problems are convex quadratic optimization problems with linear constraints, known as *quadratic programming* (QP) problems.
- One way to train a hard margin linear SVM classifier is to use a QP solver, passing it preceding parameters.
- You can also use a QP solver to solve the soft margin problem.

The Dual Problem:

- A *dual problem* is a different, yet closely related way to express a constrained optimization problem, also known as *primal problem*.
- The dual problem solution results in a lower bound to the solution of the primal problem, and sometimes it can have the same results as the primal problem.

The Dual Problem:

Equation 5-6. Dual form of the linear SVM objective

$$\underset{\alpha}{\text{minimize}} \; \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \mathbf{x}^{(j)} - \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

The Primal Problem:

Equation 5-7. From the dual solution to the primal solution

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m} \widehat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(t^{(i)} - \widehat{\mathbf{w}}^T \mathbf{x}^{(i)} \right)$$

- When the number of training instances is smaller than the number of features, the dual problem is faster to solve than the primal problem. The dual problem allows for the kernel trick to work, while the primal problem does not.
- When there is a training set with a large number of instances and features, it is best to use linear SVMs since kernelized can only use the dual problem. Dual form is too slow; the computational complexity of the primal form of the SVM is proportional to the number of training instances *m,* whereas the computational complexity of the dual form is proportional to a number between m_2 and m_3.

Kernelized SVM:

- A *kernel* is a function that computes the dot product $\phi(a)_T \phi(b)$ only with the original vectors *a* and *b,* it does not have to know about the transformation ϕ.
- A kernel expresses a measure of similarity between vectors, and is a shortcut that helps us do certain calculation faster which otherwise would involve computations in higher dimensional space.
- Kernels help make the data linearly separable for SVM, more specifically, *the feature vector ϕ (x) makes the data linearly separable*. Kernel is to make the calculation process faster and easier, especially when the feature vector ϕ is of very high dimension.

Online SVMs:

- With linear SVM classifiers, one can use Gradient Descent to minimize the cost function, derived from the primal problem, however do note that it converges more slowly than the methods based from the Quadratic Problem.
- The first sum in the cost function will push the model to have a small weight vector *w*, which leads to a larger margin. The second sum will calculate the total of all the margin violations.
    - An instance's margin violations = 1, if located off the street, on the correct side, otherwise if it proportional to the distance to the correct side of the street.
- Minimizing this term allows the model to lower the margin violations as much as possible.

Chapter 6: Decision Trees

- Decision Trees are multipurpose ML algorithms that can perform classification, regression, as well as multioutput tasks. They use powerful algorithms that can fit complex datasets.
    - Multioutput classification is a generalization of multilabel classification where each label can be multiclass, such as having more than two possible values.
    - An example of multioutput classification is building a system that removes noise from images. It will take the input as a noisy digital image, and will aim to output a clean digital image as an array of pixel intensities (similar to MNIST images from Ch. 3).
    - The multioutput classifier's output is multilabel (one label per pixel), and each label can have multiple values, therefore it's a good example of multioutput classification.

- Decision Trees are an important part of Random Forests, which are very powerful ML algorithms.

Training and Visualizing a Decision Tree:

- To build a Decision Tree, train the DecisionTreeClassifier, then visualize the trained decision tree with the *export_graphviz()* method to output a graph definition .dot file.
- The .dot file can be converted to a PDF or PNG using the *dot* command-line tool from the *graphviz* package.

Making Predictions:

- Decision Tree: a flow-chart like structure, where each *internal (non-leaf) node* represents a test on an attribute, each *branch* represents the outcome of the test, and each *leaf (or terminal) node* holds a class label.
- *Root node*: topmost node of the tree, with a depth of 0
- A node's samples attribute counts how many training instances it applies to.
- A node's value attribute reveals how many training instances of each class the node applies to.
- A node's *gini* attribute measures its *impurity* – a "pure" node (gini = 0) if all training instances it applies to belong to the same class.
- Decision Trees are relatively easy to understand, and such models are referred to as *white box models*.
- Random Forests or Neural Networks are referred to as *black box models*, where they make excellent predictions, and calculations are easy to verify, however it is difficult to explain why the predictions were made.

Example of a Decision Tree (using the Iris flower dataset):

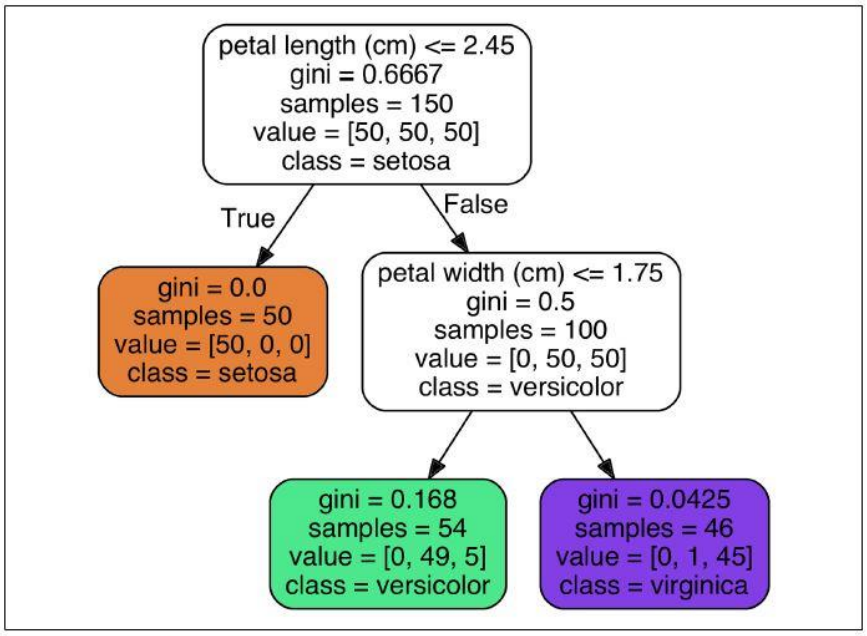

Figure 6-1. Iris Decision Tree

Estimating Class Probabilities:

- A Decision Tree can estimate the probability that an instance belongs to a specific class *k*, by traversing the tree to find the leaf node for this instance, and then returning the ratio of training instances of class *k* in the node.

The CART Training Algorithm:

- *CART (Classification and Regression Tree)* algorithm is used by Scikit-Learn to train Decision Trees (known as "growing" trees).
- The algorithm first splits the training set into two subset using a single feature, *k*, and a threshold t_k.
- *k* and t_k are chosen by searching for the pair (k, t_k) that produces the purest subsets (weighted by their size).
- After the training set is split in two, the algorithm splits the subsets with the same logic, then the following subsets, etc. It will stop recursing once the maximum depth is reached, or if it can't, then it will find a split that reduces impurity.

Computational Complexity:

- To make predictions, one has to traverse the Decision Tree from the root to a leaf.
- Decision trees are usually balanced, therefore traversing it requires going through approximately $O(log_2(m))$ nodes.
- Each node only needs the value of one feature to be checked, so the complexity is $O(log_2(m))$, independent of the number of features, which allows predictions to be very fast, despite dealing with large training sets.
- A binary Decision Tree will be balances at the end of training, with one leaf per training instance if it is trained without restrictions.
- The training algorithm though compares all features (or less if *max_features* is used) on all sample for each node. Therefore the training complexity = $O(n \times m\ log(m))$.
- For small training sets (less than a few thousand), Scikit-Learn speeds training be presorting the data, however this does slow the training for larger training sets.

Gini Impurity or Entropy:

- The default is the *Gini impurity measure*, however you can use the *entropy impurity measure* by setting the hyperparameter to entropy.
    - Gini impurity measure formula:

Equation 6-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^{2}$$

    - $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node

- Entropy comes from thermodynamics, and refers to the level of molecular disorder.
    - Entropy reaches zero when molecules are still and well ordered.
    - It has a variety of domains; one is called Shannon's information theory: the measurement of the average information content of a message – entropy is zero when all measures are identical.
    - In ML, entropy is used as an impurity measure. A set's entropy is zero when it contains instances of only one class.
- Gini impurity and Entropy impurity both lead to similar decision trees.
- A node's Gini impurity is usually less than its parent's; due to the CART algorithm's cost function (splits the node so that it lowers the weighted sum of its children's Gini impurities).
- A node can however have a higher Gini impurity than its parent, if this gain is compensated for by another decrease in another child's impurity.
- Gini impurity is slightly faster to compute, so it's a good default, however, when both measures differ, Gini impurity is likely to isolate the most frequent class in its own branch of the tree, while the entropy impurity produces slightly more balanced trees in this case.

Regularization Hyperparameters:

- Decision trees make limited assumptions about the training data (in contrast with linear models, which assume the data is linear).
- *Nonparametric model:* when left unconstrained, the tree structure will adapt to fit very closely to the training data, likely overfitting it. Nonparametric models have no parameters, since the parameters are not determined prior to training, the model can freely stick to the data.
- *Parametric model:* a model such as a linear model that has a predetermined number of parameters, therefore it has a limited degree of freedom, which lowers the risk of overfitting (but increases the chance of underfitting).
- To avoid overfitting the training data, restrict the decision tree's freedom during training, this process is regularization.
- Decision Trees are not concerned if the training data is scaled or centered; therefore scaling is pointless when a Decision Tree underfits the training set.
- Regularization hyperparameters depend on the algorithm used, but the maximum depth of the tree can be restricted.
    - Scikit-Learn controls this with the *max_depth* hyperparameter (default is *None*, meaning unlimited).
    - Reducing *max_depth* can regularize the model and lower the risk of overfitting
- The DecisionTreeClassifier has other parameters that restrict the shape of the tree:
    - *Min_samples_split* – the minimum number of samples a node must have before it can be split
    - *Min_weight_fraction_leaf* - the minimum number of samples a node must have before it can be split, expressed as a fractionof the total number of weighted instances
    - *Max_leaf_nodes* – maximum number of leaf nodes
    - *Max_features* – maximum number of features that are evaluated for splitting at each node
- Some algorithms work by training the Decision Tree without restrictions, then *pruning* (deleting) unnecessary nodes (a node's children nodes are all leaf nodes and unnecessary if the purity improvement is not *statistically significant*).
- X_2 *test*: estimates the probability that the improvement is solely the result of chance (*mull hypothesis)*
- If the P-value is > 0.05, then the node is unnecessary and its children are pruned; pruning continues till all unnecessary nodes have been taken out

Regression:

- Decision Trees can also perform regression. The regression based tree looks similar to the classification tree. The key difference is that instead of predicting a class at each node, it predicts a value.
- The predicted value for each region is the average target value of the instances in that region. The algorithm splits each region so that most training instances are as close as possible to the predicted value.
- The CART algorithm is similar, but instead of splitting the training set to reduce impurity, it splits the training set to reduce the MSE (Mean Squared Error).
- Similar to classification tree, regression trees are prone to overfitting as well. They need to be regularized to help minimize this.

Instability:

- Decision Trees are very useful however, they do have some limitations.
- Decision Trees use orthogonal decision boundaries (splits are perpendicular to an axis), which makes them sensitive to training set rotation.
    - One way to solve this problem is to use PCA, which results in a better orientation of the data.
- Another issue with Decision Trees is that they are highly sensitive to small variations in the training data.
    - One way to tackle this problem is to use Random Forests, which can reduce this instability over many trees.

Chapter 7: Ensemble Learning and Random Forests

- If you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often end up with better predictions than with the best individual predictor.
- A group of predictors is called an *ensemble*, the technique described above is called *Ensemble Learning*, and the Ensemble Learning algorithm is called an *Ensemble method.*
- *Random Forest* is an ensemble of Decision Trees, and a very powerful ML algorithm
    - You can train a group of Decision Tree classifiers, each on a different random subset of the training set.
    - To make predictions, you obtain the predictions of all individual trees, then predict the class that gets the most votes.

Voting Classifiers:

- Let's say you have already trained a few classifiers (such as Logistic Regression, KNN, etc.), with each one having about an 80% accuracy.
- If you want to make a better classifier, one way to do so is by aggregating the predictions of each classifier and predicting the class that gets the most votes, this majority-vote classifier is known as a *hard-voting classifier*.
- This voting classifier often results in higher accuracy than the best classifier in the ensemble; the ensemble can still be a *strong learner* (having high accuracy) even if each classifier is a *weak learner* (which means it only does a little better than random guessing), provided that there is enough weak learners that are sufficiently diverse.

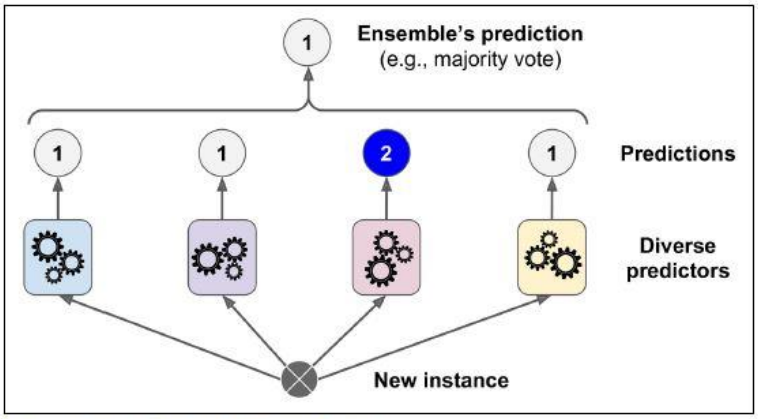

Figure 7-2. Hard voting classifier predictions

- Ensemble methods work best when the predictions are as independent from each other as possible. One way to get diverse classifiers is to train them using different algorithms, so that this increases the chance that they will make very different types of errors, thereby improving the ensemble's accuracy.
- *Soft voting:* predicting the class with the highest probability, averaged over all the individual classifiers
- Soft voting often performs better than hard voting because it gives more weight to highly confident votes.
    - To implement soft voting, set voting = "soft" and set the probability hyperparameter to "true" (ensuring the SVC class uses cross-validation to estimate class probabilities)

Bagging and Pasting:

- Another way to get a diverse set of classifiers is to use the same training algorithm for every predictor, but train them on different random subsets of the training set.
    - *Bagging:* when sampling with the above method is performed *with* replacement
    - *Pasting:* when sampling with the above method is performed *without* replacement
- Bagging and Pasting allow training instances to be sample many times across multiple predictors, but only bagging allows the training instances to be sample several times for the same predictor.
- After all the predictors are trained, the ensemble can make a prediction for a new instance by aggregating the predictions of all the predictors.
- Every individual predictor has a higher bias than if it was trained on the original training set, however aggregation helps reduce the bias and variance.
- Typically the net result is that the ensemble has a similar bias, but a lower variance than a single predictor trained on the original training set.
- Another benefit for bagging and pasting is that predictions can be made in a parallel fashion. So this means that these two methods are popular because of their flexibility to be scaled easily.

Bagging and Pasting in Scikit-Learn:

- Scikit utilizes an easy API for both bagging and pasting, using the *BaggingClassifier* class (use *BaggingRegressor* for regression).
    - If you want to use pasting instead of bagging, set *bootstrap* = False
    - *n_jobs* hyperparameter defines the number of CPU cores to use for training and predictions
- *BaggingClassifier* automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (which is seen with Decision Tree classifiers).
- In the diagram below, the ensemble's predictions will likely generalize much better than the single Decision Tree's predictions.
    - The ensemble has a comparable bias, but a smaller variance. It makes a similar number of errors on the training set, but has a less irregular decision boundary.

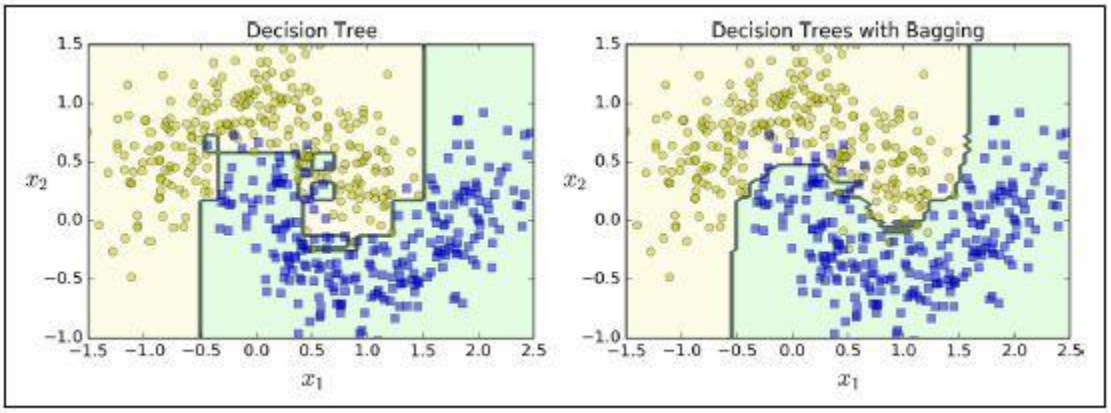

Figure 7-5. A single Decision Tree versus a bagging ensemble of 500 trees

- *Bootstrapping* introduces some more diversity in the subsets for training of each predictor, therefore bagging ends up with a slightly higher bias than pasting, but this also results in the predictors being less correlated, so the ensemble's variance is reduced. Therefore, bagging usually leads to better models.

Out-of-Bag Evaluation:

- In Bagging, some instances may be sampled several times for any predictor, while others will not be sampled at all.
- With the BaggingClassifier, only 63% of the training instances on average are sampled for each predictor.
- The leftover 37% training instances that are not sampled are called *out-of-bag* (oob) instances.
- You can include the oob instances by evaluating the ensemble itself by averaging out the oob evaluations for each predictor.

Random Patches and Random Subspaces:

- *Random Patches method:* when you sample both the training instances and features
- *Random Subspace method:* when you keep all the training instances and only sample features
    - It is an ensemble learning method that tries to reduce the correlation between estimators in the ensemble by training them on random samples of features instead of the entire feature set
    - When sampling features, it results in more predictor diversity, and you gain a bit more bias, but a lower variance

Random Forests:

- A Random Forest is an ensemble of Decision Trees, that are normally trained with the Bagging method, and usually set the *max_samples* as the size of the training set.
- A more optimal way to do this is use the *RandomForestClassifier*, and use *RandomForestRegressor* for regression.
- RandomForestClassifier has almost all of the hyperparameters of a DecisionTreeClassifier (to control how trees are grown), along with all the hyperparameters of a BaggingClassifier (to control the ensemble).
- The Random Forest algorithm has extra randomness, so instead of looking for the very best feature when splitting a node, it looks for the best feature among a random subset of features.
    - This gives a greater tree diversity, which gets a higher bias, but a lower variance, and results in an overall better model.

Extra-Trees:

- You can make trees more random by using random thresholds for each feature instead of searching for the best possible thresholds.
    - A forest of extremely random trees is known as an *Extremely Randomized Trees* (*Extra-Trees)* ensemble
    - Extra-Trees are faster to train than regular Ransom Forests because finding the best possible threshold for each feature at every node is a very time draining task for growing a tree

Feature Importance:

- Random Forests have an easy method to measure the relative importance of each feature.
- Scikit-Learn measures a feature's importance by seeing how much of the nodes that use that particular feature reduce impurity on the average, spanning across all trees in the forest.
    - This is a weighted average, where each node's weight is equal to the number of training samples that are associated with it.
    - Scikit-Learn calculates this score automatically for each feature after training, then it scales the results, making the sum of all importances equal to 1.

Boosting:

- *Boosting:* any Ensemble method that combines multiple weak learners into a strong learner
    - Most boosting methods will train predictors sequentially, each one will try to correct the one before it

AdaBoost:

- *AdaBoost* (Adaptive Boosting) is one of the popular forms of Boosting along with Gradient Boosting.
- *AdaBoost:* allows for the new predictor to correct its predecessor by paying more attention to training instances that the predecessor underfitted. This results in new predictors focusing more on the hard cases.
- Example of AdaBoost:
    - A first base classifier is trained to make predictions on the training set.
    - The relative weight of misclassified training instances is then increased.
    - A second classifier is then trained using the updated weights and then makes predictions on the training set again, and the weights are updated, and it repeats.
- The AdaBoost learning technique has some similarities with Gradient Descent, however instead of altering a single predictor's parameters to minimize a cost function, it adds predictors to ensemble, gradually making it better.
- After all the predictors are trained, the ensemble makes predictions similar to Bagging or Pasting, except that the predictors have different weights depending on their overall accuracy on the weighted training set.
- Drawback to AdaBoost:
    - It cannot be parallelized (or partially parallelized), because each predictor can only be trained after the previous predictor has been seen and evaluated.

    - Therefore it doesn't scale as efficiently as Bagging or Pasting.
- Scikit-Learn uses a multiclass version of AdaBoost called SAMME (Stagewise Additive Modeling using a Multiclass Exponential loss function), and it is the same as AdaBoost when there are only 2 classes.
    - A variant of it is SAMME.R (R stands for "real") and it used probabilities instead of predictions, resulting in better performance overall

Gradient Boosting:

- *Gradient Boosting* is similar to AdaBoost, and works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, unlike altering the instance weights at every iteration like AdaBoost, Gradient Boosting tries to fit the new predictor to the *residual errors* made by the previous predictor.
- Gradient Boosted Regression Trees (GBRT) is used for regression tasks, and a simple way to way to train GBRT ensembles is by using Scikit-Learn's *GradientBoostingRegressor* class.
    - The *learning_rate* hyperparameter scales the contribution amount of each tree, so if it is set to a low value (i.e. 0.1), you will need more trees in the ensemble to fit the training set, however the prediction will generalize better. This is known as *shrinkage*, a regularization technique.
- The figure below shows two GBRTs trained with a low learning rate. The left figure does not have enough trees to fit the training set, the right figure has too many trees and overfits the training set.

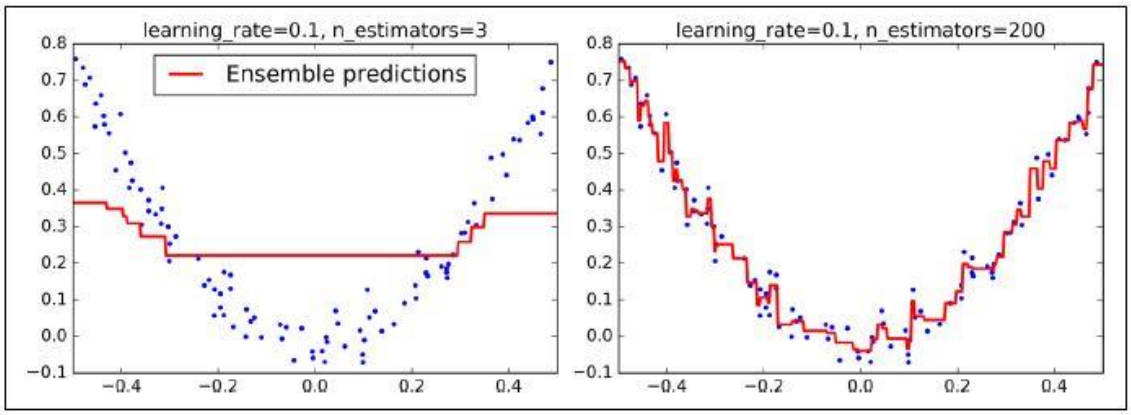

Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)

- To find the optimal number of trees, use Early Stopping (stop training as soon as the validation error reaches a minimum).
- You can also implement early stopping by stopping the training early, instead of training a large number of trees first and then looking back to find the optimal number.

Stacking:

- *Stacking* (Stacked Generalization): train the model to perform the aggregation of the predictions of all predictors in an ensemble.
    - Each of the bottom predictors predicts a different value and then the final predictor (known as a *blender* or *meta learner*) takes the predictions as inputs and makes the final prediction.
- Training a Blender (using hold-out set approach):
    - First split the training set into two subsets:
        - The first subset is used to train the predictors in the first layer.
        - The first layers are used to make predictions on the second (held-out) set, which makes sure the predictions are "clean" because the predictors never see these instances during training.
        - Each instance in the hold-out set has three predicted values, and the Blender is trained on this new training set, learning how to predict the target value with the first layer's predictions.

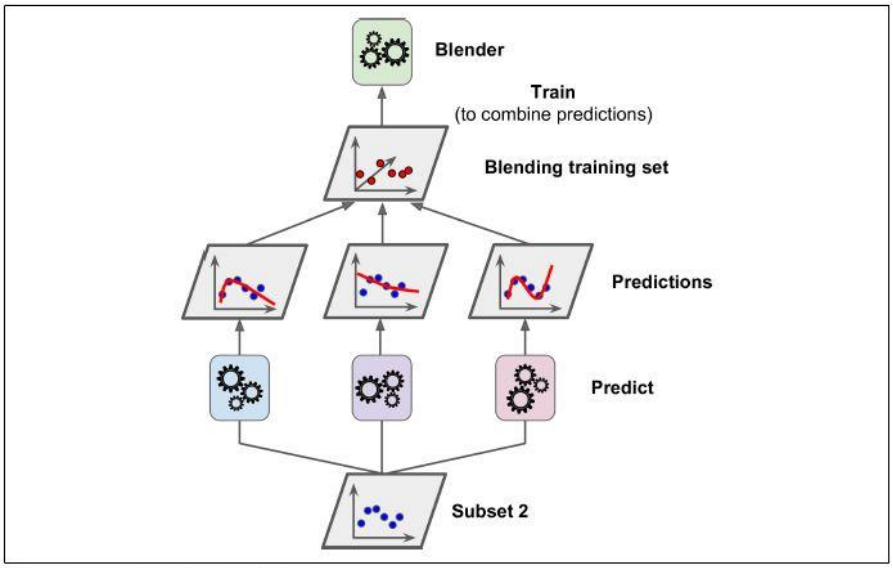

Figure 7-14. Training the blender

- You can train several different blenders by splitting the training set into three subsets.
    - The first subset is used to train the first layer.
    - The second subset is used to create the training set used to train the second layer, by using predictions from the predictors from the first layer.
    - The third subset is used to create the training set to train the third layer, by using the predictions of the predictors from the second layer.
    - After all this, we can make a prediction for a new instance by going through each layer sequentially.

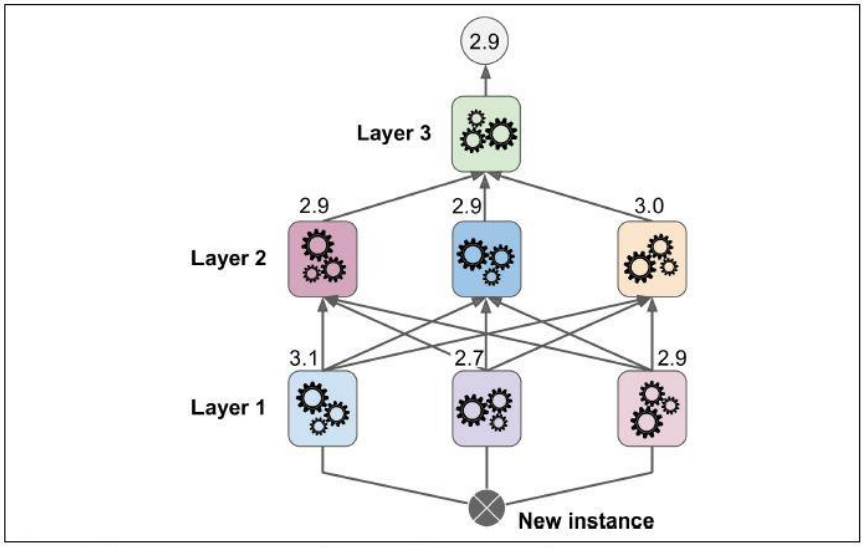

Figure 7-15. Predictions in a multilayer stacking ensemble

Chapter 8: Unsupervised Learning - Clustering

Unsupervised Learning Process:

The main uses for Unsupervised Learning include:

- Segmenting datasets by some shared attributes
- Detecting anomalies that do not fit to any group
- Simplify datasets by aggregating variables with similar attributes

The main goal is to study the intrinsic (and commonly hidden) structure of the data with unsupervised learning.

There are two main types of problems that unsupervised learning tries to solve:

- Clustering
- Dimensionality Reduction

Different Types of Clusters:

K-means: This is a clustering technique that attempts to find a user-specified number of clusters (K), which are represented by their centroids.

Agglomerative Hierarchical Clustering: This is a clustering approach that refers to a collection of closely related clustering techniques that produce a hierarchical clustering by starting with each point as a single cluster, and then repeatedly merging the two closest clusters until a single, all-encompassing cluster remains.

K-means:

K-means is a prototype-based clustering technique that can create one-level partitioning of the data objects.

K-means contains centroids, which are the mean of a group of points, and it typically applied to objects in a continuous n-dimensional space.

The Basic K-means Algorithm:

The K-Means algorithms aim to find and group in classes the data points that have high similarity between them. In the terms of the algorithm, this similarity is understood as the opposite of the distance between data points. The closer the data points are the more similar and more likely to belong to the same cluster.

For the algorithm, first choose *K* initial centroids, where K is a user-specified parameter, K = the number of clusters desired. Each point is then assigned to the closest centroid, and each collection of points assigned to a centroid is a cluster. The centroid of each cluster is then updated based on the points assigned to the cluster. You then repeat the assignment and update steps until no point changes clusters, or equivalently, until the centroids remain the same.

K-Means Hyperparameters:

- Number of clusters: The number of clusters and centroids to generate.
- Maximum iterations: Of the algorithm for a single run.
- Number initial: The number of times the algorithm will be run with different centroid seeds. The final result will be the best output of the number defined of consecutives runs, in terms of inertia.

Assigning Points to the Closest Centroid:

To assign a point to the closest centroid, we need a proximity measure that quantifies the notions of "closest" for the specific data under consideration. Euclidean (L_2) distance is often used for data points in Euclidean space. However, there are many types of proximity measures that are appropriate for a specific type of data. The K-means algorithm repeatedly calculates the similarity of each point to each centroid and the process is relatively simple, however in certain cases, such as when the data is in low-dimensional Euclidean space, it's possible to avoid computing many of the similarities, and speed up the processing.

Centroids and Objective Functions:

The K-means algorithm re-computes the centroid of each cluster, since the centroid can vary, depending on the proximity measure for the data and the goal of clustering.

The purpose of clustering is usually expressed by an objective function that depends on the proximities of the points to one another or to the cluster centroids, such as minimizing the squared distance of each point to its closest centroid.

Once you have a specified a proximity measure and an objective function, the centroid that we should choose can be mathematically calculated, but for the purpose of these notes, we'll explain it in a non-mathematical form.

Data in Euclidean Space:

For data whose proximity measure is Euclidean distance, you use the *sum of squared error* (SSE), also known as scatter, for the objective function, which measures the quality of a clustering. Cluster inertia is the name given to the Sum of Squared Errors within the clustering context. You calculate the error of each data point (its Euclidean distance to the closest centroid), and then compute the total sum of the squared errors.

When two different sets of clusters are produced by two different runs of the K-means, we prefer the one with the smallest squared error because this means that the prototypes (centroids) of the clustering are a better representation of the points in their cluster.

Choosing Initial Centroids:

Choosing the proper initial centroids is the key step of the basic K-means; a common approach is to choose the initial centroids randomly, but the resulting clusters are often not as good. Limitations of centroids are that the initial selection of the centroids may be poor, leading to suboptimal clusters.

One way to reduce this limitation is to perform multiple runs, each with a different set of randomly chosen initial centroids, and then select the set of clusters with the minimum SSE. Although this is a simple strategy, it may not always work well depending on the data set and the number of clusters that you seek. K-means attempts to minimize the total squared error.

Issues with K-Means:

Handling Empty Outliers:

One of the issues with basic K-means is that empty clusters can be obtained if there are no points are allocated to a cluster during the initialization phase. If this occurs, then you need a strategy to replace the centroid, otherwise it'll result in a high squared error. One method is to choose a centroid far away from any current centroid, which can help eliminate the point with the high squared error.

Another way to choose a replacement centroid from the cluster with the highest SSE, this will lead to the cluster being split and it will reduce the overall SSE for the clustering. If there are many empty clusters, then this process can be repeated multiple times.

Outliers:

When evaluating clusters based on squared error, outliers can have a big influence on the clusters that are found. When outliers are present, the resulting cluster centroids are not as representative of the data and they should be, and as a result, the SSE will be higher too. When and when not to eliminate outliers, depends on the dataset.

When clustering is used for data compression, every point needs to be clustered, but for something like financial analysis, outliers might be interesting to observe. Ways to find outliers can be identified in preprocessing, such as keeping track of the SSE contributed by each point, and then eliminate those points with high contributions, over multiple runs. Also we may look at eliminating small clusters since more often than not, they represent groups of outliers.

Reducing the SSE with Postprocessing:

A clear way to reduce the SSE is to find more clusters, is to use larger K. However in many cases, we want to improve the SSE, but not increase the number of clusters.

Two ways to decrease the total SSE but increasing the number of clusters are:

- *Split a cluster:* The cluster with the largest SSE is usually picked, but we can also split the cluster with the largest standard deviation for one particular attribute.

- *Introduce a new cluster centroid:* Usually the point that is farthest from any cluster center is chosen. We can determine this if we keep track of the SSE contributed from each point. Another way to is randomly choose from all points or from the points with the highest SSE.

Two ways to decrease the number of clusters, while minimizing the increase of total SSE are:

- *Disperse a cluster:* Remove the centroid that corresponds to the cluster and reassign the points to other clusters.
- *Merge two clusters:* Clusters with the closest centroids are usually chosen, or merge the two clusters that results in the smallest increase in total SSE.

Bisecting K-means:

This is an extension of the basic K-means algorithm that doe the following: to obtain K clusters, split the set of all points into two clusters, select one of these clusters to split, and so on, until K clusters have been produced.

There are multiple ways to choose a cluster to split. We can pick the largest cluster at each step, choose the one with the largest SSE, or use criteria that are based on both size and SSE.

Strengths and Weaknesses:

K-means is simple and can be used for a variety of data types. It is also efficient even though multiple runs are performed. Some other methods like bisecting K-means are more efficient and less susceptible to initialization problems. K-means is not suitable for all type of data sets though. It's not the best choice for clusters with different sizes and densities.

Agglomerative Hierarchical Clustering:

Agglomerative: Start with the points as individual clusters and, at each step, merge the closest pair of clusters. This approach needs a notion of cluster proximity. Agglomerative hierarchical clustering techniques are more commonly used. A hierarchical clustering is often displayed graphically using a tree-like diagram, a *dendrogram*, which displays both the cluster-subcluster relationships and the order in which the clusters were merged (agglomerative view).

Many agglomerative hierarchical clustering techniques are variations on a single approach: starting with individual points as clusters, successively merge the two closest clusters until only one cluster remains.

Defining Proximity between Clusters:

Cluster proximity is usually defined with a particular type of cluster in mind. Many agglomerative hierarchical clustering techniques, such as min, max, and group average, come from graph-based view of clusters. Min defines the cluster proximity as the proximity between the closest two points that are in different clusters. Max takes the proximity between the farthest two points in different clusters to be the cluster proximity. The group average makes the cluster proximity the average pairwise proximities of all pairs of points from different clusters.

Centroid methods calculate the proximity between two clusters by calculating the distance between the centroids of clusters. Centroid methods, unlike other hierarchical clustering techniques, have the possibility of *inversions*. Two clusters that are merged may be more similar (less distant) than the pair of clusters that were merged in a previous step.

Issues with Hierarchical Clustering:

Some limitations of hierarchical clustering include:

1. There is no mathematical objective for Hierarchical clustering.
2. All the approaches to calculate the similarity between clusters has its own disadvantages.
3. High space and time complexity for Hierarchical clustering. Hence this clustering algorithm cannot be used when we have huge data.

Lack of a Global Objective Function:

Agglomerative hierarchical clustering cannot be viewed as a global optimizing an objective function, instead it is used to decide criteria locally, at each step, which clusters should be merged. This approach does not have problems with local minima or difficulties choosing initial points.

Ability to Handle Different Cluster Sizes:

In agglomerative clustering, there are two ways to treat the relative sizes of the pairs of the clusters. The first one is *weighted*, which treats all clusters equally, and *unweighted*, takes the number of points in each cluster into account. Treating clusters of unequal size equally gives different weights to the points in different clusters, while taking the cluster size into account gives points in different clusters the same weight. In general, unweighted approaches are preferred unless there is a reason to believe that individual points should have different weights, such as maybe classes of objects have been unevenly sampled.

Merging Decisions are Final:

Agglomerative hierarchical clustering algorithms usually make good local decisions about combining two clusters since they can use information about the pairwise similarity of all points. However, once two clusters are merged, it cannot be undone at a later time. This approach prevents a local optimization criterion from becoming a global criterion.

There are some techniques that try to overcome the limitation that merges are final. One approach tries to fix the hierarchical clustering by moving branches of the tree around, to try to improve the global objective function. Another way is to use a partitional clustering technique such as K-means to create many small clusters, and then perform hierarchical clustering using these small clusters as the starting point.

Agglomerative clustering algorithms are typically used because the underlying application requires a hierarchy. There algorithms do produce better-quality clusters, however, they are expensive in terms of computational and storage requirements. Since all merges are final, that also causes trouble for noisy, high-dimensional data. These two issues can be resolved to a certain degree by first partially clustering the data using another technique, such as K-means.

Chapter 9: Dimensionality Reduction

- ML problems typically involve a large amount of features for each training instance, and in return, this makes training extremely slow, making it hard to come up with a good solution. This whole obstacle is referred to as the *curse of dimensionality.*
- Thankfully for real-world problems, you can significantly lower the number of features.
    - Example: For the MNIST images dataset (a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau)
        1. You can eliminate the pixels in the border of the images which are primarily white, without losing any valuable information.
        2. You can also merge neighboring pixels into one if they are highly correlated, without losing much valuable information.
- Reducing dimensionality will speed up training, but does lose some information and make the system perform a little worse while making pipelines harder to maintain.
    - Therefore to avoid this, try to train with the system with the original data before going to dimensionality reduction
- Dimensional reduction is also useful for data visualization, making it possible to plot a condensed view of a high-dimensional training set as a graph.
- There are two main approaches for dimensionality reduction:
    - Projection
    - Manifold Learning
- Three common dimensionality reduction techniques are:
    - PCA
    - Kernel PCA
    - LLE

The Curse of Dimensionality:

- An increase in the number of dimensions of a dataset results in more entries in the vector of features that represents each observation.
- Each new dimension adds a non-negative term to the sum, so that the distance increases with the number of dimensions for distinct vectors.
- One solution to the curse of dimensionality is to increase the size of the training set to reach a sufficient density of training instances.
    - However in reality, the number of training instances needed to reach a given density will grow exponentially with the number of dimensions.

Main Approaches for Dimensionality Reduction:

Projection:

- In reality, training instance are not uniformly spread across all dimensions
- Many features are almost constant, while others are highly correlate, and as a result, all training instances lie within a much lower-dimensional subspace of the high-dimensional subspace.
- In the diagram below, notice how almost all the training instances lie close to a plane, this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space.

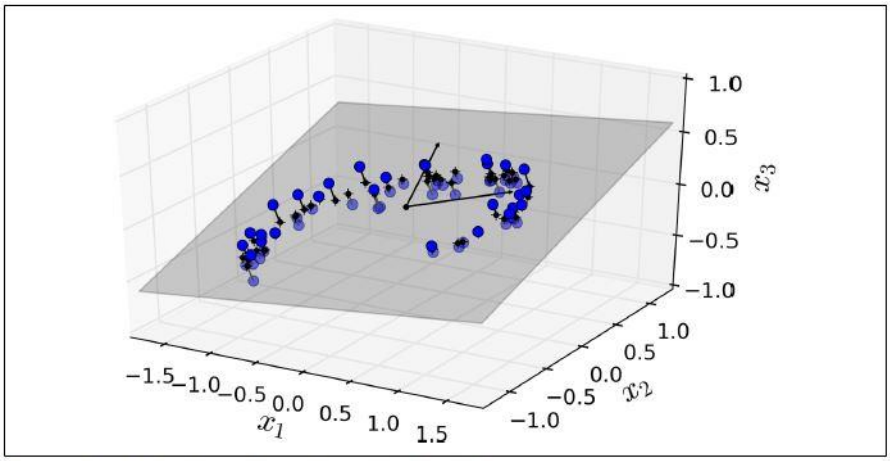

Figure 8-2. A 3D dataset lying close to a 2D subspace

- If you then project all the training instances perpendicularly onto this subspace (shown with the short lines connecting the instances to the plane), you end up with a new 2D dataset as shown in the figure below.

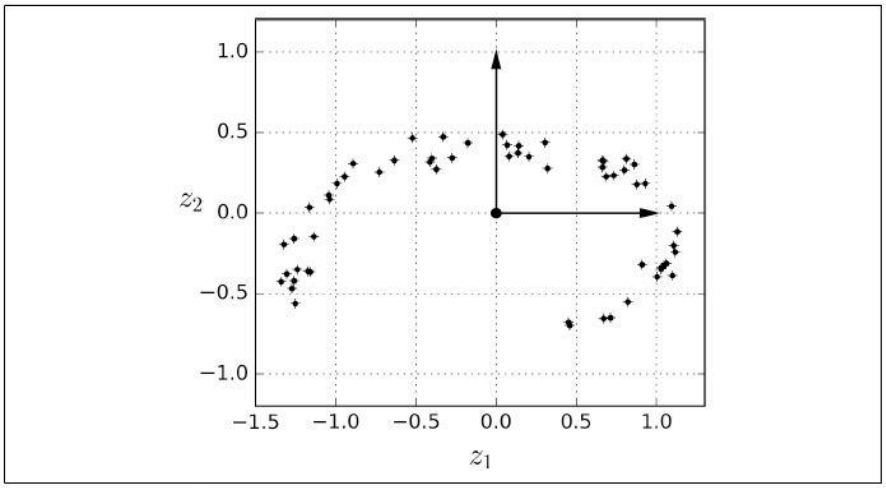

Figure 8-3. The new 2D dataset after projection

- Projection is not always the best for dimensionality reduction because the subspace may twist and turn, like in the Swiss Roll example data set.
- The best way to correct this is to not squash the data to flatten it, but rather unroll the data.

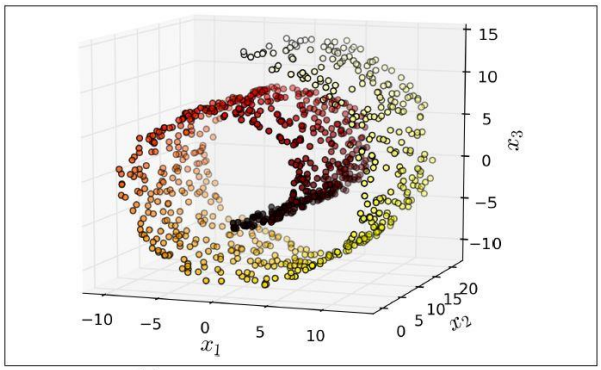

Figure 8-4. Swiss roll dataset

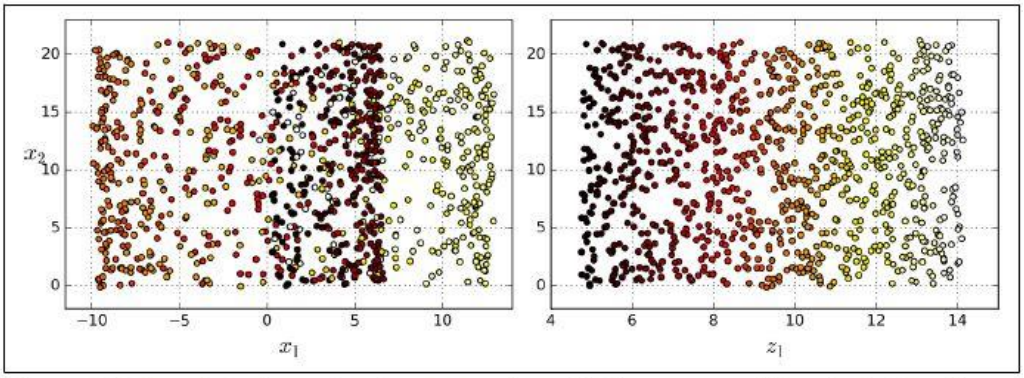

Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

Manifold Learning:

- *2D Manifold:* a 2D shape that can be bent and twisted in a higher dimensional-space (the Swiss Roll is an example of a 2D manifold)
- *D*-dimensional manifold is a part of an *n*-dimensional space (where $d < n$) that locally resembles a *d*-dimensional hyperplane
- Manifold Learning: dimensionality reduction algorithms based on modeling the manifold where the training instances lie
    - It depends on the manifold assumption/hypothesis, which agrees that high-dimensional datasets lie close to a much lower-dimensional manifold
- The manifold assumption assumes that the classification or regression task will be simpler if expressed in the lower-dimensional space of the manifold; however this is not always true.
- If you reduce the dimensionality of the training set before training a model, it typically speeds up the training, but it does not always result in a better or simpler solution, that all depends on the dataset itself.

PCA:

- *Principal Component Analysis* (PCA): a popular dimensionality reduction algorithm that identifies the hyperplane that lies closest to the data, and then projects the data onto it

Preserving the Variance:

- Before projecting the training set onto a lower-dimensional hyperplane, you need to first choose the right hyperplane.
- Select the axis that preserves the maximum amount of variance, since it'll result in the least amount of information loss compared to other projections.
    - This axis will be minimizing the mean squared distance between the original dataset and its project onto that axis.

Principal Components:

- PCA identifies the axis that accounts for the largest amount of variance in the training set.
    - It also identifies a second axis that is orthogonal to the first one, which accounts for the largest amount of remaining variance
    - In higher-dimensional datasets, PCA will continue and find another axis, orthogonal to both previous axes, and will continue on with as many axes as the number of dimensions in the dataset.
- The unit vector that defines the i^{th} axis is known as the i^{th} *principal component* (PC)
- To find the principal components of a training set, you use a standard matrix factorization technique called *Singular Value Decomposition* (SVD):
    - This will decompose the training set matrix X into the matrix multiplication of the three matrices $\mathbf{U}\ \Sigma\ \mathbf{V}^T$, where **V** contains all the principal components.

Equation 8-1. Principal components matrix

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}$$

Projecting Down to d Dimensions:

- After you identify all the principal components, next reduce the dimensionality of the dataset down to *d* dimensions by projecting it onto the hyperplane defined by the first *d* principal components.
- By selecting this hyperplane, it selects the projection that will preserve as much variance as possible.
- To project the training set onto the hyperplane, compute the matrix multiplication of the training set matrix **X** by the matrix $\mathbf{W}_d$ (matrix containing the first *d* principal components)

Using Scikit-Learn:

- Scikit-Learn's PCA class implements PCA by using SVD decomposition

Explained Variance Ratio:

- *Explained Variance Ratio*: the proportion of the dataset's variance that lies along the axis of each principal component

Choosing the Right Number of Dimensions:

- Rather than figuring out the number of dimensions to reduce down to, it's better to choose the number of dimensions that add up to a large portion of the variance.
- You can compute PCA without reducing dimensionality and then compute the minimum number of dimensions required to preserve a specified percentage of the training set's variance.
- Another way to compute PCA which is even better is to indicate the ratio of variance you want to preserve instead of specifying the number of principal components to preserve.
- An additional thing to do is to plot the explained variance as a function of the number of dimensions.

PCA for Compression:

- After dimensionality reduction, the training set will take up a significantly lower amount of space.
- After applying PCA to a dataset, it can be compressed, while still having its variance preserved, and now that a good compression ratio has been achieved, it can speed up a classification algorithm.
- You can also decompress a reduced dataset by using the inverse transformation of the PCA projection – it won't return the original data since the projection loses a bit of information, but it will be very close to the original set.
- *Reconstruction Error:* the mean squared distance between the original data and the reconstructed data (compressed and then decompressed)

Randomized PCA:

- Scikit-Learn utilizes a stochastic algorithm called *Randomized PCA* which quickly finds an estimate of the first *d* principal components.
- *Randomized PCA* is automatically used if *m* or *n* is greater than 500 and *d* is less than 80% of *m* or *n,* otherwise the full SVD approach is used.

Incremental PCA:

- One issue with PCA is that they require the whole training set to fit in memory in order for the algorithm to run.
- *Incremental PCA* helps solve this problem, by allowing you to split the training set into mini-batches and apply an IPCA algorithm one mini-batch at a time.
    - This is helpful for large training sets

Kernel PCA:

- Kernel trick: a mathematical technique that maps instances into a very high-dimensional space (known as *feature space*), which enables nonlinear classification and regression with SVM.
- A linear decision boundary in the high-dimensional feature space is linked to a complex nonlinear decision boundary in the *original space*.
- The same concept can be applied to PCA, which allows it to perform complex nonlinear projections for dimensionality reduction, known as *Kernel PCA* (kPCA).
    - kPCA is good at preserving clusters of instances after projection, or unrolling datasets that lie close to a twisted manifold.

Selecting a Kernel and Tuning Hyperparameters:

- Since kPCA is an unsupervised learning algorithm, there is no obvious performance measure to help select the best kernel and hyperparameter values.
- On the contrary, dimensionality reduction is usually a step used for supervised learning tasks like classification, so you can use grid search to pick the kernel and hyperparameters that lead to the best performance on that task.
- One way to do kPCA is to make a two-step pipeline:
    - First reduce dimensionality to two dimensions with kPCA
    - Then apply Logistic Regression for classification
    - Then use GridSearchCV to find the best kernel and gamma values for kPCA to get the best classification accuracy
- Another way to do kPCA, but entirely unsupervised, is to select the kernel and hyperparameters that results with the lowest reconstruction error.
    - Achieving a low reconstruction error is not easy, so one work around is:
        - Find a point in the original space that maps close enough to the reconstructed point – *pre-image*
        - Next, measure the squared distance of the pre-image to the original distance
        - Then select the kernel and hyperparameters that minimize the reconstruction pre-image error
        - To perform this reconstruction, one way is to train a supervised regression model, with the projected instances as the training set and the original instances as the target
            - Then use GridSearchCV to find the kernel and hyperparameters that minimize the pre-image reconstruction error

LLE:

- *Locally Linear Embedding* (LLE): a powerful *nonlinear dimensionality reduction* (NLDR) technique
  - It is a Manifold Learning technique that does not depend on projections
  - It first measures how each training instance linearly relates to its closest neighbors and then looks for a low-dimensional representation of the training set where these local relationships are best preserved
  - It is useful for unrolling twisted manifolds, especially ones with a lot of noise
- LLE work process:
  - For each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its *k* closest neighbors (in the preceding code *k* = 10)
  - Then it will try to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors
    - More specifically, it finds the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}^{(i)}$ and $\Sigma^{m}_{j=1} w_{i,j} \mathbf{x}^{(j)}$ is as small as possible, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not one of the *k* closest neighbors of $\mathbf{x}^{(i)}$

Equation 8-4. LLE step 1: linearly modeling local relationships

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} \mathbf{x}^{(j)} \right)^2$$

$$\text{subject to} \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1, 2, \cdots, m \end{cases}$$

  - For the next equation shown below, it shows an unconstrained optimization problem.
    - Instead of keeping the instances fixed and finding optimal weights, the opposite is done:
      - Keep the weights fixed and find the optimal position of the instances' images in the low-dimensional space
      - In the equation, note that Z is the matrix containing all $z^{(i)}$

Equation 8-5. LLE step 2: reducing dimensionality while preserving relationships

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Other Dimensionality Reduction Techniques:

- Some other popular dimensionality reduction techniques that are used (many with Scikit-Learn) include but not limited to:
    - *Multidimensional Scaling* (MDS): reduces dimensionality while trying to preserve the distances between the instances
    - *Isomap*: creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances* between the instances.
    - *t-Distributed Stochastic Neighbor Embedding* (t-SNE): reduces dimensionality while trying to keep similar instances close and dissimilar instances apart.
    - *Linear Discriminant Analysis* (LDA): it's a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data.