1. What is the underlying concept of Support Vector Machines?

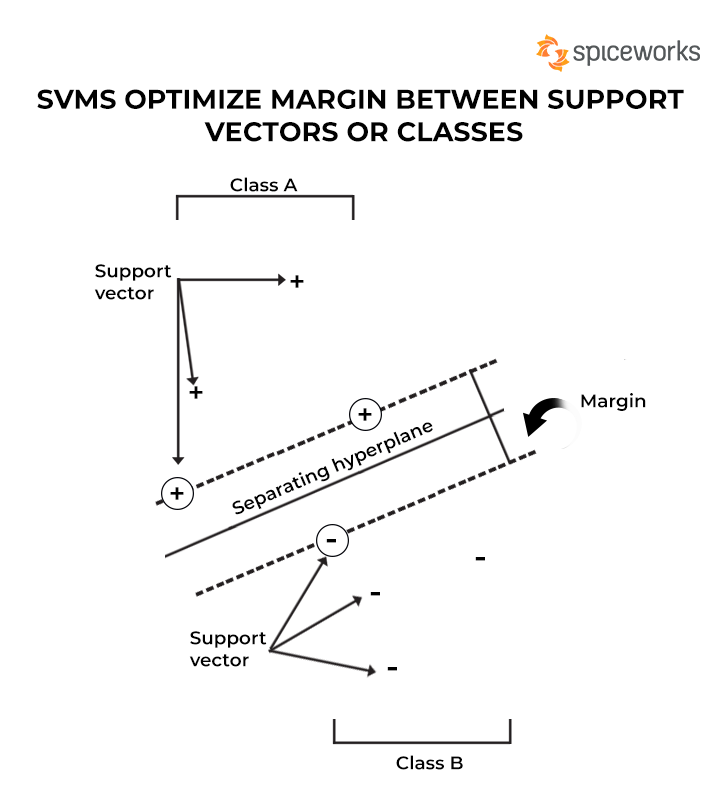
A support vector machine (SVM) is defined as a machine learning algorithm that uses supervised learning models to solve complex classification, regression, and outlier detection problems by performing optimal data transformations that determine boundaries between data points based on predefined classes, labels, or outputs. This article explains the fundamentals of SVMs, their working, types, and a few real-world examples.

What Is a Support Vector Machine?

**A support vector machine (SVM) is a machine learning algorithm that uses supervised learning models to solve complex classification, regression, and outlier detection problems by performing optimal data transformations that determine boundaries between data points based on predefined classes, labels, or outputs. SVMs are widely adopted across disciplines such as healthcare, natural language processing, signal processing applications, and speech & image recognition fields.**

Technically, the primary objective of the SVM algorithm is to identify a hyperplane that distinguishably segregates the data points of different classes. The hyperplane is localized in such a manner that the largest margin separates the classes under consideration.

The support vector representation is shown in the figure below:



**SVMs Optimize Margin Between Support Vectors or Classes**

As seen in the above figure, the margin refers to the maximum width of the slice that runs parallel to the hyperplane without any internal support vectors. Such hyperplanes are easier to define for linearly separable problems; however, for real-life problems or scenarios, the SVM algorithm tries to maximize the margin between the support vectors, thereby giving rise to incorrect classifications for smaller sections of data points.

SVMs are potentially designed for binary classification problems. However, with the rise in computationally intensive multiclass problems, several binary classifiers are constructed and combined to formulate SVMs that can implement such multiclass classifications through binary means.

In the mathematical context, an SVM refers to a set of[ML algorithms](https://www.spiceworks.com/tech/artificial-intelligence/articles/top-ml-algorithms/) that use kernel methods to transform data features by employing kernel functions. Kernel functions rely on the process of mapping complex datasets to higher dimensions in a manner that makes data point separation easier. The function simplifies the data boundaries for non-linear problems by adding higher dimensions to map complex data points.

While introducing additional dimensions, the data is not entirely transformed as it can act as a computationally taxing process. This technique is usually referred to as the kernel trick, wherein data transformation into higher dimensions is achieved efficiently and inexpensively.

The idea behind the SVM algorithm was first captured in 1963 by Vladimir N. Vapnik and Alexey Ya. Chervonenkis. Since then, SVMs have gained enough popularity as they have continued to have wide-scale implications across several areas, including the protein sorting process, text categorization, facial recognition, autonomous cars, robotic systems, and so on.

**See More:**[**What Is a Neural Network? Definition, Working, Types, and Applications in 2022**](https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-a-neural-network/)

How Does a Support Vector Machine Work?

The working of a support vector machine can be better understood through an example. Let’s assume we have red and black labels with the features denoted by x and y. We intend to have a classifier for these tags that classifies data into either the red or black category.

Let’s plot the labeled data on an x-y plane, as below:

A typical SVM separates these data points into red and black tags using the hyperplane, which is a two-dimensional line in this case. The hyperplane denotes the decision boundary line, wherein data points fall under the red or black category.

A hyperplane is defined as a line that tends to widen the margins between the two closest tags or labels (red and black). The distance of the hyperplane to the most immediate label is the largest, making the data classification easier.

The above scenario is applicable for linearly separable data. However, for non-linear data, a simple straight line cannot separate the distinct data points.

Here’s an example of non-linear complex dataset data:

The above dataset reveals that a single hyperplane is not sufficient to separate the involved labels or tags. However, here, the vectors are visibly distinct, making segregating them easier.

For data classification, you need to add another dimension to the feature space. For linear data discussed until this point, two dimensions of x and y were sufficient. In this case, we add a z-dimension to better classify the data points. Moreover, for convenience, let’s use the equation for a circle, z = x² + y².

With the third dimension, the slice of feature space along the z-direction looks like this:

Now, with three dimensions, the hyperplane, in this case, runs parallel to the x-direction at a particular value of z; let’s consider it as z=1.

The remaining data points are further mapped back to two dimensions.

The above figure reveals the boundary for data points along features x, y, and z along a circle of the circumference with radii of 1 unit that segregates two labels of tags via the SVM.

Let’s consider another method of visualizing data points in three dimensions for separating two tags (two different colored tennis balls in this case). Consider the balls lying on a 2D plane surface. Now, if we lift the surface upward, all the tennis balls are distributed in the air. The two differently colored balls may separate in the air at one point in this process. While this occurs, you can use or place the surface between two segregated sets of balls.

In this entire process, the act of ‘lifting’ the 2D surface refers to the event of mapping data into higher dimensions, which is technically referred to as ‘kernelling’, as mentioned earlier. In this way, complex data points can be separated with the help of more dimensions. The concept highlighted here is that the data points continue to get mapped into higher dimensions until a hyperplane is identified that shows a clear separation between the data points.

The figure below gives the 3D visualization of the above use case:

**See More:**[**Narrow AI vs. General AI vs. Super AI: Key Comparisons**](https://www.spiceworks.com/tech/artificial-intelligence/articles/narrow-general-super-ai-difference/)

Types of Support Vector Machines

Support vector machines are broadly classified into two types: simple or linear SVM and kernel or non-linear SVM.

1. Simple or linear SVM

A linear SVM refers to the SVM type used for classifying linearly separable data. This implies that when a dataset can be segregated into categories or classes with the help of a single straight line, it is termed a linear SVM, and the data is referred to as linearly distinct or separable. Moreover, the classifier that classifies such data is termed a linear SVM classifier.

A simple SVM is typically used to address classification and[regression analysis](https://www.spiceworks.com/tech/artificial-intelligence/articles/linear-regression-vs-logistic-regression/) problems.

2. Kernel or non-linear SVM

Non-linear data that cannot be segregated into distinct categories with the help of a straight line is classified using a kernel or non-linear SVM. Here, the classifier is referred to as a non-linear classifier. The classification can be performed with a non-linear data type by adding features into higher dimensions rather than relying on 2D space. Here, the newly added features fit a hyperplane that helps easily separate classes or categories.

Kernel SVMs are typically used to handle optimization problems that have multiple variables.

**See More:**[**What is Sentiment Analysis? Definition, Tools, and Applications**](https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-sentiment-analysis/)

Examples of Support Vector Machines

SVMs rely on supervised learning methods to classify unknown data into known categories. These find applications in diverse fields.

Here, we’ll look at some of the top real-world examples of SVMs:

Addressing the geo-sounding problem

The geo-sounding problem is one of the widespread use cases for SVMs, wherein the process is employed to track the planet’s layered structure. This entails solving the inversion problems where the observations or results of the issues are used to factor in the variables or parameters that produced them.

In the process, linear function and support vector algorithmic models separate the electromagnetic data. Moreover, linear programming practices are employed while developing the supervised models in this case. As the problem size is considerably small, the dimension size is inevitably tiny, which accounts for mapping the planet’s structure.

1. Assessing seismic liquefaction potential

Soil liquefaction is a significant concern when events such as earthquakes occur. Assessing its potential is crucial while designing any civil infrastructure. SVMs play a key role in determining the occurrence and non-occurrence of such liquefaction aspects. Technically, SVMs handle two tests: SPT (Standard Penetration Test) and CPT (Cone Penetration Test), which use field data to adjudicate the seismic status.

Moreover, SVMs are used to develop models that involve multiple variables, such as soil factors and liquefaction parameters, to determine the ground surface strength. It is believed that SVMs achieve an accuracy of close to 96-97% for such applications.

3. Protein remote homology detection

Protein remote homology is a field of computational biology where proteins are categorized into structural and functional parameters depending on the sequence of amino acids when sequence identification is seemingly difficult. SVMs play a key role in remote homology, with kernel functions determining the commonalities between protein sequences.

Thus, SVMs play a defining role in computational biology.

4. Data classification

SVMs are known to solve complex mathematical problems. However, smooth SVMs are preferred for data classification purposes, wherein smoothing techni

ques that reduce the data outliers and make the pattern identifiable are used.

Thus, for optimization problems, smooth SVMs use algorithms such as the Newton-Armijo algorithm to handle larger datasets that conventional SVMs cannot. Smooth SVM types typically explore math properties such as strong convexity for more straightforward data classification, even with non-linear data.

5. Facial detection & expression classification

SVMs classify facial structures vs. non-facial ones. The training data uses two classes of face entity (denoted by +1) and non-face entity (denoted as -1) and n\*n pixels to distinguish between face and non-face structures. Further, each pixel is analyzed, and the features from each one are extracted that denote face and non-face characters. Finally, the process creates a square decision boundary around facial

structures based on pixel intensity and classifies the resultant images.

Moreover, SVMs are also used for facial expression classification, which includes expressions denoted as happy, sad, angry, surprised, and so on.

6. Surface texture classification

In the current scenario, SVMs are used for the classification of images of surfaces. Implying that the images clicked of surfaces can be fed into SVMs to determine the texture of surfaces in those images and classify them as smooth or gritty surfaces.

7. Text categorization & handwriting recognition

Text categorization refers to classifying data into predefined categories. For example, news articles contain politics, business, the stock market, or sports. Similarly, one can segregate emails into spam, non-spam, junk, and others.

Technically, each article or document is assigned a score, which is then compared to a predefined threshold value. The article is classified into its respective category depending on the evaluated score.

For handwriting recognition examples, the dataset containing passages that different individuals write is supplied to SVMs. Typically, SVM classifiers are trained with sample data initially and are later used to classify handwriting based on score values. Subsequently, SVMs are also used to segregate writings by humans and computers.

8. Speech recognition

In speech recognition examples, words from speeches are individually picked

These methods collect audio data, feed it to SVMs and then train the models for[speech recognition](https://www.spiceworks.com/tech/artificial-intelligence/articles/speech-recognition-software/).

9. Stenography detection

With SVMs, you can determine whether any digital image is tampered with, contaminated, or pure. Such examples are helpful when handling security-related matters for organizations or government agencies, as it is easier to encrypt and embed data as a watermark in high-resolution images.

Such images contain more pixels; hence, it can be challenging to spot hidden or watermarked messages. However, one solution is to separate each pixel and store data in different datasets that SVMs can later analyze.

10. Cancer detection

Medical professionals, researchers, and scientists worldwide have been toiling hard to find a solution that can effectively detect cancer in its early stages. Today, several[AI](https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-ai/) and ML tools are being deployed for the same. For example, in January 2020, Google developed an AI tool that helps in early breast cancer detection and reduces false positives and negatives.

In such examples, SVMs can be employed, wherein cancerous images can be supplied as input. SVM algorithms can analyze them, train the models, and eventually categorize the images that reveal malign or benign cancer features.

1. What is the concept of a support vector?

# Suport Vector Machine (SVM) Algorithm

Support Vector Machine (SVM) is a powerful machine learning algorithm used for linear or nonlinear classification, regression, and even outlier detection tasks. SVMs can be used for a variety of tasks, such as text classification, image classification, spam detection, handwriting identification, gene expression analysis, face detection, and anomaly detection. SVMs are adaptable and efficient in a variety of applications because they can manage high-dimensional data and nonlinear relationships.

SVM algorithms are very effective as we try to find the maximum separating hyperplane between the different classes available in the target feature.

## Support Vector Machine

Support Vector Machine (SVM) is a [supervised machine learning](https://www.geeksforgeeks.org/supervised-unsupervised-learning/) algorithm used for both classification and regression. Though we say regression problems as well it’s best suited for classification. The main objective of the SVM algorithm is to find the optimal [hyperplane](https://www.geeksforgeeks.org/separating-hyperplanes-in-svm/) in an N-dimensional space that can separate the data points in different classes in the feature space. The hyperplane tries that the margin between the closest points of different classes should be as maximum as possible. The dimension of the hyperplane depends upon the number of features. If the number of input features is two, then the hyperplane is just a line. If the number of input features is three, then the hyperplane becomes a 2-D plane. It becomes difficult to imagine when the number of features exceeds three.

Let’s consider two independent variables x1, x2, and one dependent variable which is either a blue circle or a red circle.



*Linearly Separable Data points*

From the figure above it’s very clear that there are multiple lines (our hyperplane here is a line because we are considering only two input features x1, x2) that segregate our data points or do a classification between red and blue circles. So how do we choose the best line or in general the best hyperplane that segregates our data points?

### How does SVM work?

One reasonable choice as the best hyperplane is the one that represents the largest separation or margin between the two classes.



*Multiple hyperplanes separate the data from two classes*

So we choose the hyperplane whose distance from it to the nearest data point on each side is maximized. If such a hyperplane exists it is known as the **maximum-margin hyperplane/hard margin**. So from the above figure, we choose L2. Let’s consider a scenario like shown below



*Selecting hyperplane for data with outlier*

Here we have one blue ball in the boundary of the red ball. So how does SVM classify the data? It’s simple! The blue ball in the boundary of red ones is an outlier of blue balls. The SVM algorithm has the characteristics to ignore the outlier and finds the best hyperplane that maximizes the margin. SVM is robust to outliers.



*Hyperplane which is the most optimized one*

So in this type of data point what SVM does is, finds the maximum margin as done with previous data sets along with that it adds a penalty each time a point crosses the margin. So the margins in these types of cases are called **soft margins**. When there is a soft margin to the data set, the SVM tries to minimize (1/margin+∧(∑penalty)). Hinge loss is a commonly used penalty. If no violations no hinge loss.If violations hinge loss proportional to the distance of violation.

Till now, we were talking about linearly separable data(the group of blue balls and red balls are separable by a straight line/linear line). What to do if data are not linearly separable?



*Original 1D dataset for classification*

Say, our data is shown in the figure above. SVM solves this by creating a new variable using a **kernel**. We call a point xi on the line and we create a new variable yi as a function of distance from origin o.so if we plot this we get something like as shown below



*Mapping 1D data to 2D to become able to separate the two classes*

In this case, the new variable y is created as a function of distance from the origin. A non-linear function that creates a new variable is referred to as a kernel.

### Support Vector Machine Terminology

1. **Hyperplane:**Hyperplane is the decision boundary that is used to separate the data points of different classes in a feature space. In the case of linear classifications, it will be a linear equation i.e. wx+b = 0.
2. **Support Vectors:**Support vectors are the closest data points to the hyperplane, which makes a critical role in deciding the hyperplane and margin.
3. **Margin**: Margin is the distance between the support vector and hyperplane. The main objective of the support vector machine algorithm is to maximize the margin.  The wider margin indicates better classification performance.
4. **Kernel**: Kernel is the mathematical function, which is used in SVM to map the original input data points into high-dimensional feature spaces, so, that the hyperplane can be easily found out even if the data points are not linearly separable in the original input space. Some of the common kernel functions are linear, polynomial, radial basis function(RBF), and sigmoid.
5. **Hard Margin:** The maximum-margin hyperplane or the hard margin hyperplane is a hyperplane that properly separates the data points of different categories without any misclassifications.
6. **Soft Margin:**When the data is not perfectly separable or contains outliers, SVM permits a soft margin technique. Each data point has a slack variable introduced by the soft-margin SVM formulation, which softens the strict margin requirement and permits certain misclassifications or violations. It discovers a compromise between increasing the margin and reducing violations.
7. **C:**Margin maximisation and misclassification fines are balanced by the regularisation parameter C in SVM. The penalty for going over the margin or misclassifying data items is decided by it. A stricter penalty is imposed with a greater value of C, which results in a smaller margin and perhaps fewer misclassifications.
8. **Hinge Loss:** A typical loss function in SVMs is hinge loss. It punishes incorrect classifications or margin violations. The objective function in SVM is frequently formed by combining it with the regularisation term.
9. **Dual Problem:** A dual Problem of the optimisation problem that requires locating the Lagrange multipliers related to the support vectors can be used to solve SVM. The dual formulation enables the use of kernel tricks and more effective computing.

3. When using SVMs, why is it necessary to scale the inputs?

According to the documentation of the [StandardScaler](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) object in scikit-learn:

For instance many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger that others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

I should scale my features before classification. Is there any easy way to show why I should do this? References to scientific articles would be even better. I already found [one](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.2658&rep=rep1&type=pdf) but there are probably many other.

4. When an SVM classifier classifies a case, can it output a confidence score? What about a percentage chance?

I have two classes (say 1 and 0), and want to build a classifier. It is possible to use artificial neural networks (ANN) or any "real" classifying method such as SVM or Random Forest. In case of ANN, one can easily estimates confidence level of classification. For example, if we have binary task (with outputs as 0 or 1), and ANN results for some sample is 0.92, one can suppose that ANN "sure" in classification to 1 class. Alternatively, if ANN outputs 0.52, it is considered as unsteady classification to 1 flass.

5. Should you train a model on a training set with millions of instances and hundreds of features using the primal or dual form of the SVM problem?

This question applies only to linear SVMs since kernelized can only use the dual form. The computational complexity of the primal form of the SVM problem is proportional to the number of training instances m, while the computational complexity of the dual form is proportional to a number between m2 and m3. So if there are millions of instances, you should definitely use the primal form, because the dual form will be much too slow.

6. Let's say you've used an RBF kernel to train an SVM classifier, but it appears to underfit the training collection. Is it better to raise or lower (gamma)? What about the letter C?

If an SVM classifier trained with an RBF kernel underfits the training set, there might be too much regularization. To decrease it, you need to increase gamma or C (or both).

7. To solve the soft margin linear SVM classifier problem with an off-the-shelf QP solver, how should the QP parameters (H, f, A, and b) be set?

Let’s call the QP parameters for the hard-margin problem H′, f′, A′ and b′. The QP parameters for the soft-margin problem have m additional parameters (np = n + 1 + m) and m additional constraints (nc = 2m). They can be defined like so:

* H is equal to H′, plus m columns of 0s on the right and m rows of 0s at the bottom
* f is equal to f′ with m additional elements, all equal to the value of the hyperparameter C.
* b is equal to b′ with m additional elements, all equal to 0.
* A is equal to A′, with an extra m × m identity matrix Im appended to the right,

8. On a linearly separable dataset, train a LinearSVC. Then, using the same dataset, train an SVC and an SGDClassifier. See if you can get them to make a model that is similar to yours.

A Support Vector Machine (SVM) is a powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex small- or medium-sized datasets.

This chapter will explain the core concepts of SVMs, how to use them, and how they work.

# Linear SVM Classification

The fundamental idea behind SVMs is best explained with some pictures. [Figure 5-1](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#large_margin_classification_plot) shows part of the iris dataset that was introduced at the end of [Chapter 4](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch04.html#linear_models_chapter). The two classes can clearly be separated easily with a straight line (they are linearly separable). The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line is so bad that it does not even separate the classes properly. The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances. In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also stays as far away from the closest training instances as possible. You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called large margin classification.

Notice that adding more training instances “off the street” will not affect the decision boundary at all: it is fully determined (or “supported”) by the instances located on the edge of the street. These instances are called the support vectors (they are circled in [Figure 5-1](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#large_margin_classification_plot)).

###### **WARNING**

SVMs are sensitive to the feature scales, as you can see in [Figure 5-2](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#sensitivity_to_feature_scales_plot): in the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using Scikit-Learn’s StandardScaler), the decision boundary in the right plot looks much better.

## ****Soft Margin Classification****

If we strictly impose that all instances must be off the street and on the right side, this is called hard margin classification. There are two main issues with hard margin classification. First, it only works if the data is linearly separable. Second, it is sensitive to outliers. [Figure 5-3](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#sensitivity_to_outliers_plot) shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin; on the right, the decision boundary ends up very different from the one we saw in [Figure 5-1](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#large_margin_classification_plot) without the outlier, and it will probably not generalize as well.

To avoid these issues, use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the margin violations (i.e., instances that end up in the middle of the street or even on the wrong side). This is called soft margin classification.

When creating an SVM model using Scikit-Learn, we can specify a number of hyperparameters. C is one of those hyperparameters. If we set it to a low value, then we end up with the model on the left of [Figure 5-4](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#regularization_plot). With a high value, we get the model on the right. Margin violations are bad. It’s usually better to have few of them. However, in this case the model on the left has a lot of margin violations but will probably generalize better.

###### **TIP**

If your SVM model is overfitting, you can try regularizing it by reducing C.

The following Scikit-Learn code loads the iris dataset, scales the features, and then trains a linear SVM model (using the LinearSVC class with C=1 and the hinge loss function, described shortly) to detect Iris virginica flowers:

**import numpy as np**

**from sklearn import datasets**

**from sklearn.pipeline import Pipeline**

**from sklearn.preprocessing import StandardScaler**

**from sklearn.svm import LinearSVC**

**iris = datasets.load\_iris()**

**X = iris["data"][:, (2, 3)]  # petal length, petal width**

**y = (iris["target"] == 2).astype(np.float64)  # Iris virginica**

**svm\_clf = Pipeline([**

**("scaler", StandardScaler()),**

**("linear\_svc", LinearSVC(C=1, loss="hinge")),**

**])**

**svm\_clf.fit(X, y)**

The resulting model is represented on the left in [Figure 5-4](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#regularization_plot).

Then, as usual, you can use the model to make predictions:

**>>> svm\_clf.predict([[5.5, 1.7]])**

**array([1.])**

###### **NOTE**

Unlike Logistic Regression classifiers, SVM classifiers do not output probabilities for each class.

Instead of using the LinearSVC class, we could use the SVC class with a linear kernel. When creating the SVC model, we would write SVC(kernel="linear", C=1). Or we could use the SGDClassifier class, with SGDClassifier(loss="hinge", alpha=1/(m\*C)). This applies regular Stochastic Gradient Descent (see [Chapter 4](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch04.html#linear_models_chapter)) to train a linear SVM classifier. It does not converge as fast as the LinearSVC class, but it can be useful to handle online classification tasks or huge datasets that do not fit in memory (out-of-core training).

###### **TIP**

The LinearSVC class regularizes the bias term, so you should center the training set first by subtracting its mean. This is automatic if you scale the data using the StandardScaler. Also make sure you set the loss hyperparameter to "hinge", as it is not the default value. Finally, for better performance, you should set the dual hyperparameter to False, unless there are more features than training instances (we will discuss duality later in the chapter).

# Nonlinear SVM Classification

Although linear SVM classifiers are efficient and work surprisingly well in many cases, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features (as you did in [Chapter 4](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch04.html#linear_models_chapter)); in some cases this can result in a linearly separable dataset. Consider the left plot in [Figure 5-5](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#higher_dimensions_plot): it represents a simple dataset with just one feature, x**1**. This dataset is not linearly separable, as you can see. But if you add a second feature x**2** = (x**1**)**2**, the resulting 2D dataset is perfectly linearly separable.

To implement this idea using Scikit-Learn, create a Pipeline containing a PolynomialFeatures transformer (discussed in [“Polynomial Regression”](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch04.html#polynomial_regression)), followed by a StandardScaler and a LinearSVC. Let’s test this on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving half circles (see [Figure 5-6](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#moons_polynomial_svc_plot)). You can generate this dataset using the make\_moons() function:

**from sklearn.datasets import make\_moons**

**from sklearn.pipeline import Pipeline**

**from sklearn.preprocessing import PolynomialFeatures**

**X, y = make\_moons(n\_samples=100, noise=0.15)**

**polynomial\_svm\_clf = Pipeline([**

**("poly\_features", PolynomialFeatures(degree=3)),**

**("scaler", StandardScaler()),**

**("svm\_clf", LinearSVC(C=10, loss="hinge"))**

**])**

**polynomial\_svm\_clf.fit(X, y)**

## ****Polynomial Kernel****

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs). That said, at a low polynomial degree, this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the kernel trick (explained in a moment). The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with very high-degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features because you don’t actually add any features. This trick is implemented by the SVC class. Let’s test it on the moons dataset:

**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)**

This code trains an SVM classifier using a third-degree polynomial kernel. It is represented on the left in [Figure 5-7](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#moons_kernelized_polynomial_svc_plot). On the right is another SVM classifier using a 10th-degree polynomial kernel. Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter coef0 controls how much the model is influenced by high-degree polynomials versus low-degree polynomials.

###### **TIP**

A common approach to finding the right hyperparameter values is to use grid search (see [Chapter 2](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#project_chapter)). It is often faster to first do a very coarse grid search, then a finer grid search around the best values found. Having a good sense of what each hyperparameter actually does can also help you search in the right part of the hyperparameter space.

## ****Similarity Features****

Another technique to tackle nonlinear problems is to add features computed using a similarity function, which measures how much each instance resembles a particular landmark. For example, let’s take the 1D dataset discussed earlier and add two landmarks to it at x**1** = –2 and x**1** = 1 (see the left plot in [Figure 5-8](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#kernel_method_plot)). Next, let’s define the similarity function to be the Gaussian Radial Basis Function (RBF) with γ = 0.3 (see [Equation 5-1](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#grbf_function)).

##### **Equation 5-1. Gaussian RBF**

**��(�,ℓ)=exp(-��-ℓ2)**

This is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark). Now we are ready to compute the new features. For example, let’s look at the instance x**1** = –1: it is located at a distance of 1 from the first landmark and 2 from the second landmark. Therefore its new features are x**2** = exp(–0.3 × 1**2**) ≈ 0.74 and x**3** = exp(–0.3 × 2**2**) ≈ 0.30. The plot on the right in [Figure 5-8](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#kernel_method_plot) shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset. Doing that creates many dimensions and thus increases the chances that the transformed training set will be linearly separable. The downside is that a training set with m instances and n features gets transformed into a training set with m instances and m features (assuming you drop the original features). If your training set is very large, you end up with an equally large number of features.

## ****Gaussian RBF Kernel****

Just like the polynomial features method, the similarity features method can be useful with any Machine Learning algorithm, but it may be computationally expensive to compute all the additional features, especially on large training sets. Once again the kernel trick does its SVM magic, making it possible to obtain a similar result as if you had added many similarity features. Let’s try the SVC class with the Gaussian RBF kernel:

**rbf\_kernel\_svm\_clf = Pipeline([**

**("scaler", StandardScaler()),**

**("svm\_clf", SVC(kernel="rbf", gamma=5, C=0.001))**

**])**

**rbf\_kernel\_svm\_clf.fit(X, y)**

This model is represented at the bottom left in [Figure 5-9](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#moons_rbf_svc_plot). The other plots show models trained with different values of hyperparameters gamma (γ) and C. Increasing gamma makes the bell-shaped curve narrower (see the lefthand plots in [Figure 5-8](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#kernel_method_plot)). As a result, each instance’s range of influence is smaller: the decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small gamma value makes the bell-shaped curve wider: instances have a larger range of influence, and the decision boundary ends up smoother. So γ acts like a regularization hyperparameter: if your model is overfitting, you should reduce it; if it is underfitting, you should increase it (similar to the C hyperparameter).

Other kernels exist but are used much more rarely. Some kernels are specialized for specific data structures. String kernels are sometimes used when classifying text documents or DNA sequences (e.g., using the string subsequence kernel or kernels based on the Levenshtein distance).

###### **TIP**

With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first (remember that LinearSVC is much faster than SVC(kernel="linear")), especially if the training set is very large or if it has plenty of features. If the training set is not too large, you should also try the Gaussian RBF kernel; it works well in most cases. Then if you have spare time and computing power, you can experiment with a few other kernels, using cross-validation and grid search. You’d want to experiment like that especially if there are kernels specialized for your training set’s data structure.

## ****Computational Complexity****

The LinearSVC class is based on the liblinear library, which implements an [optimized algorithm](https://homl.info/13) for linear SVMs.[**1**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#idm45022164653368) It does not support the kernel trick, but it scales almost linearly with the number of training instances and the number of features. Its training time complexity is roughly O(m × n).

The algorithm takes longer if you require very high precision. This is controlled by the tolerance hyperparameter ϵ (called tol in Scikit-Learn). In most classification tasks, the default tolerance is fine.

The SVC class is based on the libsvm library, which implements [an algorithm](https://homl.info/14) that supports the kernel trick.[**2**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#idm45022164645000) The training time complexity is usually between O(m**2** × n) and O(m**3** × n). Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (e.g., hundreds of thousands of instances). This algorithm is perfect for complex small or medium-sized training sets. It scales well with the number of features, especially with sparse features (i.e., when each instance has few nonzero features). In this case, the algorithm scales roughly with the average number of nonzero features per instance. [Table 5-1](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#svm_classification_algorithm_comparison) compares Scikit-Learn’s SVM classification classes.

Table 5-1. Comparison of Scikit-Learn classes for SVM classification**ClassTime complexityOut-of-core supportScaling requiredKernel trick**

LinearSVC

O(m × n)

No

Yes

No

SGDClassifier

O(m × n)

Yes

Yes

No

SVC

O(m² × n) to O(m³ × n)

No

Yes

Yes

# SVM Regression

As mentioned earlier, the SVM algorithm is versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression. To use SVMs for regression instead of classification, the trick is to reverse the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instances off the street). The width of the street is controlled by a hyperparameter, ϵ. [Figure 5-10](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#svm_regression_plot) shows two linear SVM Regression models trained on some random linear data, one with a large margin (ϵ = 1.5) and the other with a small margin (ϵ = 0.5).

Adding more training instances within the margin does not affect the model’s predictions; thus, the model is said to be ϵ-insensitive.

You can use Scikit-Learn’s LinearSVR class to perform linear SVM Regression. The following code produces the model represented on the left in [Figure 5-10](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#svm_regression_plot) (the training data should be scaled and centered first):

**from sklearn.svm import LinearSVR**

**svm\_reg = LinearSVR(epsilon=1.5)**

**svm\_reg.fit(X, y)**

To tackle nonlinear regression tasks, you can use a kernelized SVM model. [Figure 5-11](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#svm_with_polynomial_kernel_plot) shows SVM Regression on a random quadratic training set, using a second-degree polynomial kernel. There is little regularization in the left plot (i.e., a large C value), and much more regularization in the right plot (i.e., a small C value).

The following code uses Scikit-Learn’s SVR class (which supports the kernel trick) to produce the model represented on the left in [Figure 5-11](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#svm_with_polynomial_kernel_plot):

**from sklearn.svm import SVR**

**svm\_poly\_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)**

**svm\_poly\_reg.fit(X, y)**

The SVR class is the regression equivalent of the SVC class, and the LinearSVR class is the regression equivalent of the LinearSVC class. The LinearSVR class scales linearly with the size of the training set (just like the LinearSVC class), while the SVR class gets much too slow when the training set grows large (just like the SVC class).

###### **NOTE**

SVMs can also be used for outlier detection; see Scikit-Learn’s documentation for more details.

# Under the Hood

This section explains how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. If you are just getting started with Machine Learning, you can safely skip it and go straight to the exercises at the end of this chapter, and come back later when you want to get a deeper understanding of SVMs.

First, a word about notations. In [Chapter 4](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch04.html#linear_models_chapter) we used the convention of putting all the model parameters in one vector **θ**, including the bias term θ**0** and the input feature weights θ**1** to θ***n***, and adding a bias input x**0** = 1 to all instances. In this chapter we will use a convention that is more convenient (and more common) when dealing with SVMs: the bias term will be called b, and the feature weights vector will be called **w**. No bias feature will be added to the input feature vectors.

## ****Decision Function and Predictions****

The linear SVM classifier model predicts the class of a new instance **x** by simply computing the decision function **w⊺** **x** + b = w**1** x**1** + ⋯ + w***n*** x***n*** + b. If the result is positive, the predicted class ŷ is the positive class (1), and otherwise it is the negative class (0); see [Equation 5-2](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#linear_svm_classifier_prediction).

##### **Equation 5-2. Linear SVM classifier prediction**

**�^=0if�⊺�+�<0,1if�⊺�+�≥0**

[Figure 5-12](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#iris_3D_plot) shows the decision function that corresponds to the model in the right in [Figure 5-4](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#regularization_plot): it is a 2D plane because this dataset has two features (petal width and petal length). The decision boundary is the set of points where the decision function is equal to 0: it is the intersection of two planes, which is a straight line (represented by the thick solid line).[**3**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#idm45022164414872)

The dashed lines represent the points where the decision function is equal to 1 or –1: they are parallel and at equal distance to the decision boundary, and they form a margin around it. Training a linear SVM classifier means finding the values 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****

Consider the slope of the decision function: it is equal to the norm of the weight vector, ∥ **w** ∥. If we divide this slope by 2, the points where the decision function is equal to ±1 are going to be twice as far away from the decision boundary. In other words, dividing the slope by 2 will multiply the margin by 2. This may be easier to visualize in 2D, as shown in [Figure 5-13](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#small_w_large_margin_plot). The smaller the weight vector **w**, the larger the margin.

So we want to minimize ∥ **w** ∥ to get a large margin. If we also want to avoid any margin violations (hard margin), then we need the decision function to be greater than 1 for all positive training instances and lower than –1 for negative training instances. If we define t**(*i*)** = –1 for negative instances (if y**(*i*)** = 0) and t**(*i*)** = 1 for positive instances (if y**(*i*)** = 1), then we can express this constraint as t**(*i*)**(**w⊺** **x(*i*)** + b) ≥ 1 for all instances.

We can therefore express the hard margin linear SVM classifier objective as the constrained optimization problem in [Equation 5-3](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#hard_margin_objective).

##### **Equation 5-3. Hard margin linear SVM classifier objective**

**minimize�,�12�⊺�subjectto�(�)(�⊺�(�)+�)≥1for�=1,2,⋯,�**

###### **NOTE**

We are minimizing ½ **w⊺** **w**, which is equal to ½∥ **w** ∥**2**, rather than minimizing ∥ **w** ∥. Indeed, ½∥ **w** ∥**2** has a nice, simple derivative (it is just **w**), while ∥ **w** ∥ is not differentiable at **w** = 0. Optimization algorithms work much better on differentiable functions.

To get the soft margin objective, we need to introduce a slack variable ζ**(*i*)** ≥ 0 for each instance:[**4**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#idm45022164350760) ζ**(*i*)** measures how much the i**th** instance is allowed to violate the margin. We now have two conflicting objectives: make the slack variables as small as possible to reduce the margin violations, and make ½ **w⊺** **w** as small as possible to increase the margin. This is where the C hyperparameter comes in: it allows us to define the tradeoff between these two objectives. This gives us the constrained optimization problem in [Equation 5-4](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#soft_margin_objective).

##### **Equation 5-4. Soft margin linear SVM classifier objective**

**minimize�,�,�12�⊺�+�∑�=1��(�)subjectto�(�)(�⊺�(�)+�)≥1-�(�)and�(�)≥0for�=1,2,⋯,�**

## ****Quadratic Programming****

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. Such problems are known as Quadratic Programming (QP) problems. Many off-the-shelf solvers are available to solve QP problems by using a variety of techniques that are outside the scope of this book.[**5**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#idm45022164298152)

The general problem formulation is given by [Equation 5-5](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch05.html#quadratic_programming_problem_formulation).

##### **Equation 5-5. Quadratic Programming problem**

**Minimize�12�⊺��+�⊺�subjectto��≤�where�isan��-dimensionalvector(��=numberofparameters),�isan��×��matrix,�isan��-dimensionalvector,�isan��×��matrix(��=numberofconstraints),�isan��-dimensionalvector.**

Note that the expression **A** **p** ≤ **b** defines n***c*** constraints: **p⊺** **a(*i*)** ≤ b**(*i*)** for i = 1, 2, ⋯, n***c***, where **a(*i*)** is the vector containing the elements of the i**th** row of **A** and b**(*i*)** is the i**th** element of **b**.

You can easily verify that if you set the QP parameters in the following way, you get the hard margin linear SVM classifier objective:

* n***p*** = n + 1, where n is the number of features (the +1 is for the bias term).
* n***c*** = m, where m is the number of training instances.
* **H** is the n***p*** × n***p*** identity matrix, except with a zero in the top-left cell (to ignore the bias term).
* **f** = 0, an n***p***-dimensional vector full of 0s.
* **b** = –1, an n***c***-dimensional vector full of –1s.
* **a(*i*)** = –t**(*i*)** �˙**(*i*)**, where �˙**(*i*)** is equal to **x(*i*)** with an extra bias feature �˙**0** = 1.

One way to train a hard margin linear SVM classifier is to use an off-the-shelf QP solver and pass it the preceding parameters. The resulting vector **p** will contain the bias term b = p**0** and the feature weights w***i*** = p***i*** for i = 1, 2, ⋯, n. Similarly, you can use a QP solver to solve the soft margin problem (see the exercises at the end of the chapter).

To use the kernel trick, we are going to look at a different constrained optimization problem.

9. On the MNIST dataset, train an SVM classifier. You'll need to use one-versus-the-rest to assign all 10 digits because SVM classifiers are binary classifiers. To accelerate up the process, you might want to tune the hyperparameters using small validation sets. What level of precision can you achieve?

# Linear SVM Classification

The next few code cells generate the first figures in chapter 5. The first actual code sample comes after.

**Code to generate Figure 5–1. Large margin classification**

In [2]:

**from** sklearn.svm **import** SVC

**from** sklearn **import** datasets

iris **=** datasets**.**load\_iris()

X **=** iris["data"][:, (2, 3)] *# petal length, petal width*

y **=** iris["target"]

setosa\_or\_versicolor **=** (y **==** 0) **|** (y **==** 1)

X **=** X[setosa\_or\_versicolor]

y **=** y[setosa\_or\_versicolor]

*# SVM Classifier model*

svm\_clf **=** SVC(kernel**=**"linear", C**=**float("inf"))

svm\_clf**.**fit(X, y)

*# Bad models*

x0 **=** np**.**linspace(0, 5.5, 200)

pred\_1 **=** 5**\***x0 **-** 20

pred\_2 **=** x0 **-** 1.8

pred\_3 **=** 0.1 **\*** x0 **+** 0.5

**def** plot\_svc\_decision\_boundary(svm\_clf, xmin, xmax):

w **=** svm\_clf**.**coef\_[0]

b **=** svm\_clf**.**intercept\_[0]

*# At the decision boundary, w0\*x0 + w1\*x1 + b = 0*

*# => x1 = -w0/w1 \* x0 - b/w1*

x0 **=** np**.**linspace(xmin, xmax, 200)

decision\_boundary **=** **-**w[0]**/**w[1] **\*** x0 **-** b**/**w[1]

margin **=** 1**/**w[1]

gutter\_up **=** decision\_boundary **+** margin

gutter\_down **=** decision\_boundary **-** margin

svs **=** svm\_clf**.**support\_vectors\_

plt**.**scatter(svs[:, 0], svs[:, 1], s**=**180, facecolors**=**'#FFAAAA')

plt**.**plot(x0, decision\_boundary, "k-", linewidth**=**2)

plt**.**plot(x0, gutter\_up, "k--", linewidth**=**2)

plt**.**plot(x0, gutter\_down, "k--", linewidth**=**2)

fig, axes **=** plt**.**subplots(ncols**=**2, figsize**=**(10,2.7), sharey**=True**)

plt**.**sca(axes[0])

plt**.**plot(x0, pred\_1, "g--", linewidth**=**2)

plt**.**plot(x0, pred\_2, "m-", linewidth**=**2)

plt**.**plot(x0, pred\_3, "r-", linewidth**=**2)

plt**.**plot(X[:, 0][y**==**1], X[:, 1][y**==**1], "bs", label**=**"Iris versicolor")

plt**.**plot(X[:, 0][y**==**0], X[:, 1][y**==**0], "yo", label**=**"Iris setosa")

plt**.**xlabel("Petal length", fontsize**=**14)

plt**.**ylabel("Petal width", fontsize**=**14)

plt**.**legend(loc**=**"upper left", fontsize**=**14)

plt**.**axis([0, 5.5, 0, 2])

plt**.**sca(axes[1])

plot\_svc\_decision\_boundary(svm\_clf, 0, 5.5)

plt**.**plot(X[:, 0][y**==**1], X[:, 1][y**==**1], "bs")

plt**.**plot(X[:, 0][y**==**0], X[:, 1][y**==**0], "yo")

plt**.**xlabel("Petal length", fontsize**=**14)

plt**.**axis([0, 5.5, 0, 2])

save\_fig("large\_margin\_classification\_plot")

plt**.**show()

Saving figure large\_margin\_classification\_plot

**Code to generate Figure 5–2. Sensitivity to feature scales**

In [4]:

Xs **=** np**.**array([[1, 50], [5, 20], [3, 80], [5, 60]])**.**astype(np**.**float64)

ys **=** np**.**array([0, 0, 1, 1])

svm\_clf **=** SVC(kernel**=**"linear", C**=**100)

svm\_clf**.**fit(Xs, ys)

plt**.**figure(figsize**=**(9,2.7))

plt**.**subplot(121)

plt**.**plot(Xs[:, 0][ys**==**1], Xs[:, 1][ys**==**1], "bo")

plt**.**plot(Xs[:, 0][ys**==**0], Xs[:, 1][ys**==**0], "ms")

plot\_svc\_decision\_boundary(svm\_clf, 0, 6)

plt**.**xlabel("$x\_0$", fontsize**=**20)

plt**.**ylabel("$x\_1$    ", fontsize**=**20, rotation**=**0)

plt**.**title("Unscaled", fontsize**=**16)

plt**.**axis([0, 6, 0, 90])

**from** sklearn.preprocessing **import** StandardScaler

scaler **=** StandardScaler()

X\_scaled **=** scaler**.**fit\_transform(Xs)

svm\_clf**.**fit(X\_scaled, ys)

plt**.**subplot(122)

plt**.**plot(X\_scaled[:, 0][ys**==**1], X\_scaled[:, 1][ys**==**1], "bo")

plt**.**plot(X\_scaled[:, 0][ys**==**0], X\_scaled[:, 1][ys**==**0], "ms")

plot\_svc\_decision\_boundary(svm\_clf, **-**2, 2)

plt**.**xlabel("$x'\_0$", fontsize**=**20)

plt**.**ylabel("$x'\_1$ ", fontsize**=**20, rotation**=**0)

plt**.**title("Scaled", fontsize**=**16)

plt**.**axis([**-**2, 2, **-**2, 2])

save\_fig("sensitivity\_to\_feature\_scales\_plot")

Saving figure sensitivity\_to\_feature\_scales\_plot

**Soft Margin Classification**

**Code to generate Figure 5–3. Hard margin sensitivity to outliers**

In [5]:

X\_outliers **=** np**.**array([[3.4, 1.3], [3.2, 0.8]])

y\_outliers **=** np**.**array([0, 0])

Xo1 **=** np**.**concatenate([X, X\_outliers[:1]], axis**=**0)

yo1 **=** np**.**concatenate([y, y\_outliers[:1]], axis**=**0)

Xo2 **=** np**.**concatenate([X, X\_outliers[1:]], axis**=**0)

yo2 **=** np**.**concatenate([y, y\_outliers[1:]], axis**=**0)

svm\_clf2 **=** SVC(kernel**=**"linear", C**=**10**\*\***9)

svm\_clf2**.**fit(Xo2, yo2)

fig, axes **=** plt**.**subplots(ncols**=**2, figsize**=**(10,2.7), sharey**=True**)

plt**.**sca(axes[0])

plt**.**plot(Xo1[:, 0][yo1**==**1], Xo1[:, 1][yo1**==**1], "bs")

plt**.**plot(Xo1[:, 0][yo1**==**0], Xo1[:, 1][yo1**==**0], "yo")

plt**.**text(0.3, 1.0, "Impossible!", fontsize**=**24, color**=**"red")

plt**.**xlabel("Petal length", fontsize**=**14)

plt**.**ylabel("Petal width", fontsize**=**14)

plt**.**annotate("Outlier",

xy**=**(X\_outliers[0][0], X\_outliers[0][1]),

xytext**=**(2.5, 1.7),

ha**=**"center",

arrowprops**=**dict(facecolor**=**'black', shrink**=**0.1),

fontsize**=**16,

)

plt**.**axis([0, 5.5, 0, 2])

plt**.**sca(axes[1])

plt**.**plot(Xo2[:, 0][yo2**==**1], Xo2[:, 1][yo2**==**1], "bs")

plt**.**plot(Xo2[:, 0][yo2**==**0], Xo2[:, 1][yo2**==**0], "yo")

plot\_svc\_decision\_boundary(svm\_clf2, 0, 5.5)

plt**.**xlabel("Petal length", fontsize**=**14)

plt**.**annotate("Outlier",

xy**=**(X\_outliers[1][0], X\_outliers[1][1]),

xytext**=**(3.2, 0.08),

ha**=**"center",

arrowprops**=**dict(facecolor**=**'black', shrink**=**0.1),

fontsize**=**16,

)

plt**.**axis([0, 5.5, 0, 2])

save\_fig("sensitivity\_to\_outliers\_plot")

plt**.**show()

Saving figure sensitivity\_to\_outliers\_plot

**This is the first code example in chapter 5:**

In [6]:

**import** numpy **as** np

**from** sklearn **import** datasets

**from** sklearn.pipeline **import** Pipeline

**from** sklearn.preprocessing **import** StandardScaler

**from** sklearn.svm **import** LinearSVC

iris **=** datasets**.**load\_iris()

X **=** iris["data"][:, (2, 3)] *# petal length, petal width*

y **=** (iris["target"] **==** 2)**.**astype(np**.**float64) *# Iris virginica*

svm\_clf **=** Pipeline([

("scaler", StandardScaler()),

("linear\_svc", LinearSVC(C**=**1, loss**=**"hinge", random\_state**=**42)),

])

svm\_clf**.**fit(X, y)

Out[6]:

Pipeline(steps=[('scaler', StandardScaler()),

('linear\_svc', LinearSVC(C=1, loss='hinge', random\_state=42))])

In [7]:

svm\_clf**.**predict([[5.5, 1.7]])

Out[7]:

array([1.])

**Code to generate Figure 5–4. Large margin versus fewer margin violations**

In [8]:

scaler **=** StandardScaler()

svm\_clf1 **=** LinearSVC(C**=**1, loss**=**"hinge", random\_state**=**42)

svm\_clf2 **=** LinearSVC(C**=**100, loss**=**"hinge", random\_state**=**42)

scaled\_svm\_clf1 **=** Pipeline([

("scaler", scaler),

("linear\_svc", svm\_clf1),

])

scaled\_svm\_clf2 **=** Pipeline([

("scaler", scaler),

("linear\_svc", svm\_clf2),

])

scaled\_svm\_clf1**.**fit(X, y)

scaled\_svm\_clf2**.**fit(X, y)

/Users/ageron/miniconda3/envs/tf2/lib/python3.7/site-packages/sklearn/svm/\_base.py:977: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.

"the number of iterations.", ConvergenceWarning)

Out[8]:

Pipeline(steps=[('scaler', StandardScaler()),

('linear\_svc',

LinearSVC(C=100, loss='hinge', random\_state=42))])

In [9]:

*# Convert to unscaled parameters*

b1 **=** svm\_clf1**.**decision\_function([**-**scaler**.**mean\_ **/** scaler**.**scale\_])

b2 **=** svm\_clf2**.**decision\_function([**-**scaler**.**mean\_ **/** scaler**.**scale\_])

w1 **=** svm\_clf1**.**coef\_[0] **/** scaler**.**scale\_

w2 **=** svm\_clf2**.**coef\_[0] **/** scaler**.**scale\_

svm\_clf1**.**intercept\_ **=** np**.**array([b1])

svm\_clf2**.**intercept\_ **=** np**.**array([b2])

svm\_clf1**.**coef\_ **=** np**.**array([w1])

svm\_clf2**.**coef\_ **=** np**.**array([w2])

*# Find support vectors (LinearSVC does not do this automatically)*

t **=** y **\*** 2 **-** 1

support\_vectors\_idx1 **=** (t **\*** (X**.**dot(w1) **+** b1) **<** 1)**.**ravel()

support\_vectors\_idx2 **=** (t **\*** (X**.**dot(w2) **+** b2) **<** 1)**.**ravel()

svm\_clf1**.**support\_vectors\_ **=** X[support\_vectors\_idx1]

svm\_clf2**.**support\_vectors\_ **=** X[support\_vectors\_idx2]

In [10]:

fig, axes **=** plt**.**subplots(ncols**=**2, figsize**=**(10,2.7), sharey**=True**)

plt**.**sca(axes[0])

plt**.**plot(X[:, 0][y**==**1], X[:, 1][y**==**1], "g^", label**=**"Iris virginica")

plt**.**plot(X[:, 0][y**==**0], X[:, 1][y**==**0], "bs", label**=**"Iris versicolor")

plot\_svc\_decision\_boundary(svm\_clf1, 4, 5.9)

plt**.**xlabel("Petal length", fontsize**=**14)

plt**.**ylabel("Petal width", fontsize**=**14)

plt**.**legend(loc**=**"upper left", fontsize**=**14)

plt**.**title("$C = {}$"**.**format(svm\_clf1**.**C), fontsize**=**16)

plt**.**axis([4, 5.9, 0.8, 2.8])

plt**.**sca(axes[1])

plt**.**plot(X[:, 0][y**==**1], X[:, 1][y**==**1], "g^")

plt**.**plot(X[:, 0][y**==**0], X[:, 1][y**==**0], "bs")

plot\_svc\_decision\_boundary(svm\_clf2, 4, 5.99)

plt**.**xlabel("Petal length", fontsize**=**14)

plt**.**title("$C = {}$"**.**format(svm\_clf2**.**C), fontsize**=**16)

plt**.**axis([4, 5.9, 0.8, 2.8])

save\_fig("regularization\_plot")

Saving figure regularization\_plot

**Nonlinear SVM Classification**

**Code to generate Figure 5–5. Adding features to make a dataset linearly separable**

In [11]:

X1D **=** np**.**linspace(**-**4, 4, 9)**.**reshape(**-**1, 1)

X2D **=** np**.**c\_[X1D, X1D**\*\***2]

y **=** np**.**array([0, 0, 1, 1, 1, 1, 1, 0, 0])

plt**.**figure(figsize**=**(10, 3))

plt**.**subplot(121)

plt**.**grid(**True**, which**=**'both')

plt**.**axhline(y**=**0, color**=**'k')

plt**.**plot(X1D[:, 0][y**==**0], np**.**zeros(4), "bs")

plt**.**plot(X1D[:, 0][y**==**1], np**.**zeros(5), "g^")

plt**.**gca()**.**get\_yaxis()**.**set\_ticks([])

plt**.**xlabel(r"$x\_1$", fontsize**=**20)

plt**.**axis([**-**4.5, 4.5, **-**0.2, 0.2])

plt**.**subplot(122)

plt**.**grid(**True**, which**=**'both')

plt**.**axhline(y**=**0, color**=**'k')

plt**.**axvline(x**=**0, color**=**'k')

plt**.**plot(X2D[:, 0][y**==**0], X2D[:, 1][y**==**0], "bs")

plt**.**plot(X2D[:, 0][y**==**1], X2D[:, 1][y**==**1], "g^")

plt**.**xlabel(r"$x\_1$", fontsize**=**20)

plt**.**ylabel(r"$x\_2$  ", fontsize**=**20, rotation**=**0)

plt**.**gca()**.**get\_yaxis()**.**set\_ticks([0, 4, 8, 12, 16])

plt**.**plot([**-**4.5, 4.5], [6.5, 6.5], "r--", linewidth**=**3)

plt**.**axis([**-**4.5, 4.5, **-**1, 17])

plt**.**subplots\_adjust(right**=**1)

save\_fig("higher\_dimensions\_plot", tight\_layout**=False**)

plt**.**show()

Saving figure higher\_dimensions\_plot

In [12]:

**from** sklearn.datasets **import** make\_moons

X, y **=** make\_moons(n\_samples**=**100, noise**=**0.15, random\_state**=**42)

**def** plot\_dataset(X, y, axes):

plt**.**plot(X[:, 0][y**==**0], X[:, 1][y**==**0], "bs")

plt**.**plot(X[:, 0][y**==**1], X[:, 1][y**==**1], "g^")

plt**.**axis(axes)

plt**.**grid(**True**, which**=**'both')

plt**.**xlabel(r"$x\_1$", fontsize**=**20)

plt**.**ylabel(r"$x\_2$", fontsize**=**20, rotation**=**0)

plot\_dataset(X, y, [**-**1.5, 2.5, **-**1, 1.5])

plt**.**show()

**Here is second code example in the chapter:**

In [13]:

**from** sklearn.datasets **import** make\_moons

**from** sklearn.pipeline **import** Pipeline

**from** sklearn.preprocessing **import** PolynomialFeatures

polynomial\_svm\_clf **=** Pipeline([

("poly\_features", PolynomialFeatures(degree**=**3)),

("scaler", StandardScaler()),

("svm\_clf", LinearSVC(C**=**10, loss**=**"hinge", random\_state**=**42))

])

polynomial\_svm\_clf**.**fit(X, y)

/Users/ageron/miniconda3/envs/tf2/lib/python3.7/site-packages/sklearn/svm/\_base.py:977: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.

"the number of iterations.", ConvergenceWarning)

Out[13]:

Pipeline(steps=[('poly\_features', PolynomialFeatures(degree=3)),

('scaler', StandardScaler()),

('svm\_clf', LinearSVC(C=10, loss='hinge', random\_state=42))])

**Code to generate Figure 5–6. Linear SVM classifier using polynomial features**

In [14]:

**def** plot\_predictions(clf, axes):

x0s **=** np**.**linspace(axes[0], axes[1], 100)

x1s **=** np**.**linspace(axes[2], axes[3], 100)

x0, x1 **=** np**.**meshgrid(x0s, x1s)

X **=** np**.**c\_[x0**.**ravel(), x1**.**ravel()]

y\_pred **=** clf**.**predict(X)**.**reshape(x0**.**shape)

y\_decision **=** clf**.**decision\_function(X)**.**reshape(x0**.**shape)

plt**.**contourf(x0, x1, y\_pred, cmap**=**plt**.**cm**.**brg, alpha**=**0.2)

plt**.**contourf(x0, x1, y\_decision, cmap**=**plt**.**cm**.**brg, alpha**=**0.1)

plot\_predictions(polynomial\_svm\_clf, [**-**1.5, 2.5, **-**1, 1.5])

plot\_dataset(X, y, [**-**1.5, 2.5, **-**1, 1.5])

save\_fig("moons\_polynomial\_svc\_plot")

plt**.**show()

Saving figure moons\_polynomial\_svc\_plot

**Polynomial Kernel**

**Next code example:**

In [15]:

**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)

Out[15]:

Pipeline(steps=[('scaler', StandardScaler()),

('svm\_clf', SVC(C=5, coef0=1, kernel='poly'))])

**Code to generate Figure 5–7. SVM classifiers with a polynomial kernel**

In [16]:

poly100\_kernel\_svm\_clf **=** Pipeline([

("scaler", StandardScaler()),

("svm\_clf", SVC(kernel**=**"poly", degree**=**10, coef0**=**100, C**=**5))

])

poly100\_kernel\_svm\_clf**.**fit(X, y)

Out[16]:

Pipeline(steps=[('scaler', StandardScaler()),

('svm\_clf', SVC(C=5, coef0=100, degree=10, kernel='poly'))])

In [17]:

fig, axes **=** plt**.**subplots(ncols**=**2, figsize**=**(10.5, 4), sharey**=True**)

plt**.**sca(axes[0])

plot\_predictions(poly\_kernel\_svm\_clf, [**-**1.5, 2.45, **-**1, 1.5])

plot\_dataset(X, y, [**-**1.5, 2.4, **-**1, 1.5])

plt**.**title(r"$d=3, r=1, C=5$", fontsize**=**18)

plt**.**sca(axes[1])

plot\_predictions(poly100\_kernel\_svm\_clf, [**-**1.5, 2.45, **-**1, 1.5])

plot\_dataset(X, y, [**-**1.5, 2.4, **-**1, 1.5])

plt**.**title(r"$d=10, r=100, C=5$", fontsize**=**18)

plt**.**ylabel("")

save\_fig("moons\_kernelized\_polynomial\_svc\_plot")

plt**.**show()

Saving figure moons\_kernelized\_polynomial\_svc\_plot

**Similarity Features**

**Code to generate Figure 5–8. Similarity features using the Gaussian RBF**

In [18]:

**def** gaussian\_rbf(x, landmark, gamma):

**return** np**.**exp(**-**gamma **\*** np**.**linalg**.**norm(x **-** landmark, axis**=**1)**\*\***2)

gamma **=** 0.3

x1s **=** np**.**linspace(**-**4.5, 4.5, 200)**.**reshape(**-**1, 1)

x2s **=** gaussian\_rbf(x1s, **-**2, gamma)

x3s **=** gaussian\_rbf(x1s, 1, gamma)

XK **=** np**.**c\_[gaussian\_rbf(X1D, **-**2, gamma), gaussian\_rbf(X1D, 1, gamma)]

yk **=** np**.**array([0, 0, 1, 1, 1, 1, 1, 0, 0])

plt**.**figure(figsize**=**(10.5, 4))

plt**.**subplot(121)

plt**.**grid(**True**, which**=**'both')

plt**.**axhline(y**=**0, color**=**'k')

plt**.**scatter(x**=**[**-**2, 1], y**=**[0, 0], s**=**150, alpha**=**0.5, c**=**"red")

plt**.**plot(X1D[:, 0][yk**==**0], np**.**zeros(4), "bs")

plt**.**plot(X1D[:, 0][yk**==**1], np**.**zeros(5), "g^")

plt**.**plot(x1s, x2s, "g--")

plt**.**plot(x1s, x3s, "b:")

plt**.**gca()**.**get\_yaxis()**.**set\_ticks([0, 0.25, 0.5, 0.75, 1])

plt**.**xlabel(r"$x\_1$", fontsize**=**20)

plt**.**ylabel(r"Similarity", fontsize**=**14)

plt**.**annotate(r'$\mathbf{x}$',

xy**=**(X1D[3, 0], 0),

xytext**=**(**-**0.5, 0.20),

ha**=**"center",

arrowprops**=**dict(facecolor**=**'black', shrink**=**0.1),

fontsize**=**18,

)

plt**.**text(**-**2, 0.9, "$x\_2$", ha**=**"center", fontsize**=**20)

plt**.**text(1, 0.9, "$x\_3$", ha**=**"center", fontsize**=**20)

plt**.**axis([**-**4.5, 4.5, **-**0.1, 1.1])

plt**.**subplot(122)

plt**.**grid(**True**, which**=**'both')

plt**.**axhline(y**=**0, color**=**'k')

plt**.**axvline(x**=**0, color**=**'k')

plt**.**plot(XK[:, 0][yk**==**0], XK[:, 1][yk**==**0], "bs")

plt**.**plot(XK[:, 0][yk**==**1], XK[:, 1][yk**==**1], "g^")

plt**.**xlabel(r"$x\_2$", fontsize**=**20)

plt**.**ylabel(r"$x\_3$  ", fontsize**=**20, rotation**=**0)

plt**.**annotate(r'$\phi\left(\mathbf{x}\right)$',

xy**=**(XK[3, 0], XK[3, 1]),

xytext**=**(0.65, 0.50),

ha**=**"center",

arrowprops**=**dict(facecolor**=**'black', shrink**=**0.1),

fontsize**=**18,

)

plt**.**plot([**-**0.1, 1.1], [0.57, **-**0.1], "r--", linewidth**=**3)

plt**.**axis([**-**0.1, 1.1, **-**0.1, 1.1])

plt**.**subplots\_adjust(right**=**1)

save\_fig("kernel\_method\_plot")

plt**.**show()

Saving figure kernel\_method\_plot

In [19]:

x1\_example **=** X1D[3, 0]

**for** landmark **in** (**-**2, 1):

k **=** gaussian\_rbf(np**.**array([[x1\_example]]), np**.**array([[landmark]]), gamma)

print("Phi({}, {}) = {}"**.**format(x1\_example, landmark, k))

Phi(-1.0, -2) = [0.74081822]

Phi(-1.0, 1) = [0.30119421]

1. On the California housing dataset, train an SVM regressor.

# Import Libraries

In [1]:

*# Common imports*

**import** pandas **as** pd

**import** numpy **as** np

*# data visualization*

**import** matplotlib.pyplot **as** plt

**from** sklearn.model\_selection **import** train\_test\_split

*# random seed to make output stable across runs*

np**.**random**.**seed(42)

*# data visualization*

**%matplotlib** inline

**import** matplotlib

**import** matplotlib.pyplot **as** plt

plt**.**rcParams['axes.labelsize'] **=** 14

plt**.**rcParams['xtick.labelsize'] **=** 12

plt**.**rcParams['ytick.labelsize'] **=** 12

# ****California housing Dataset****

The dataset we will use is the "California Housing Prices" dataset from the statlib repository, which is based on data from the 1990 census. This dataset offers great opportunities for learning. The prediciton task for this dataset wil be to predict housing prices based on several features.

### Get the data:

In [2]:

**import** os

**import** tarfile

**from** six.moves **import** urllib

root **=** "https://raw.githubusercontent.com/ageron/handson-ml/master/"

path **=** os**.**path**.**join("datasets", "housing")

source **=** root **+** "datasets/housing/housing.tgz"

**def** fetch\_housing\_data(housing\_url**=**source, path**=**path):

**if** **not** os**.**path**.**isdir(path):

os**.**makedirs(path)

tgz\_path **=** os**.**path**.**join(path, "housing.tgz")

urllib**.**request**.**urlretrieve(housing\_url, tgz\_path)

housing\_tgz **=** tarfile**.**open(tgz\_path)

housing\_tgz**.**extractall(path**=**path)

housing\_tgz**.**close()

In [3]:

fetch\_housing\_data()

In [4]:

**import** pandas **as** pd

**def** load\_data(housing\_path**=**path):

csv **=** os**.**path**.**join(housing\_path, "housing.csv")

**return** pd**.**read\_csv(csv)

# Big Picture

You are asked to build a machine learning model that predicts the median house value for a given "district". This data has metrics like population, median income, median housing price for each block group in califorania. Block groups are smallest geographical unit for which the US Census Bureu publishes data. I will just call them "districts" for now.

Frame the problem:

Is it supervised, unsupervised or reinforcement learning ? Is it a classification task, a regression task or something different ? Our problem is clearly a supervised learning task, because you have labeled input data. It is also clearly a regression task since you have to predict a numeric value.

In [5]:

housing **=** load\_data()

housing**.**head()

Out[5]:

|  | **longitude** | **latitude** | **housing\_median\_age** | **total\_rooms** | **total\_bedrooms** | **population** | **households** | **median\_income** | **median\_house\_value** | **ocean\_proximity** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | -122.23 | 37.88 | 41.0 | 880.0 | 129.0 | 322.0 | 126.0 | 8.3252 | 452600.0 | NEAR BAY |
| **1** | -122.22 | 37.86 | 21.0 | 7099.0 | 1106.0 | 2401.0 | 1138.0 | 8.3014 | 358500.0 | NEAR BAY |
| **2** | -122.24 | 37.85 | 52.0 | 1467.0 | 190.0 | 496.0 | 177.0 | 7.2574 | 352100.0 | NEAR BAY |
| **3** | -122.25 | 37.85 | 52.0 | 1274.0 | 235.0 | 558.0 | 219.0 | 5.6431 | 341300.0 | NEAR BAY |
| **4** | -122.25 | 37.85 | 52.0 | 1627.0 | 280.0 | 565.0 | 259.0 | 3.8462 | 342200.0 | NEAR BAY |

Above you can see the top 5 rows of the dataset with the "head()" method.

Each row represents one district. The dataset has 10 attributes: longitude, latitude, housing\_median\_age, total\_rooms, total\_bedrooms, population, households, median\_income, median\_house\_value and ocean\_proximity.

In [6]:

housing**.**info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 20640 entries, 0 to 20639

Data columns (total 10 columns):

longitude 20640 non-null float64

latitude 20640 non-null float64

housing\_median\_age 20640 non-null float64

total\_rooms 20640 non-null float64

total\_bedrooms 20433 non-null float64

population 20640 non-null float64

households 20640 non-null float64

median\_income 20640 non-null float64

median\_house\_value 20640 non-null float64

ocean\_proximity 20640 non-null object

dtypes: float64(9), object(1)

memory usage: 1.6+ MB

Now you can see a brief description of the data using the "info()" method above.

Notice that there are 20,640 instances (entries) in the whole dataset. Also notice that the total\_bedrooms attribute has only 20,433 non-null values which means that 207 districts don't have this feature. You can also see that ocean\_proximity is not numerical and probably a categorical attribute. You can find out how many districts belong to each attribute by using the "value\_counts()" method, which I used below.

In [7]:

housing["ocean\_proximity"]**.**value\_counts()

Out[7]:

<1H OCEAN 9136

INLAND 6551

NEAR OCEAN 2658

NEAR BAY 2290

ISLAND 5

Name: ocean\_proximity, dtype: int64

Now I will use the "describe()" method to show a summary of the numerical attributes:

In [8]:

housing**.**describe()

Out[8]:

|  | **longitude** | **latitude** | **housing\_median\_age** | **total\_rooms** | **total\_bedrooms** | **population** | **households** | **median\_income** | **median\_house\_value** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 20640.000000 | 20640.000000 | 20640.000000 | 20640.000000 | 20433.000000 | 20640.000000 | 20640.000000 | 20640.000000 | 20640.000000 |
| **mean** | -119.569704 | 35.631861 | 28.639486 | 2635.763081 | 537.870553 | 1425.476744 | 499.539680 | 3.870671 | 206855.816909 |
| **std** | 2.003532 | 2.135952 | 12.585558 | 2181.615252 | 421.385070 | 1132.462122 | 382.329753 | 1.899822 | 115395.615874 |
| **min** | -124.350000 | 32.540000 | 1.000000 | 2.000000 | 1.000000 | 3.000000 | 1.000000 | 0.499900 | 14999.000000 |
| **25%** | -121.800000 | 33.930000 | 18.000000 | 1447.750000 | 296.000000 | 787.000000 | 280.000000 | 2.563400 | 119600.000000 |
| **50%** | -118.490000 | 34.260000 | 29.000000 | 2127.000000 | 435.000000 | 1166.000000 | 409.000000 | 3.534800 | 179700.000000 |
| **75%** | -118.010000 | 37.710000 | 37.000000 | 3148.000000 | 647.000000 | 1725.000000 | 605.000000 | 4.743250 | 264725.000000 |
| **max** | -114.310000 | 41.950000 | 52.000000 | 39320.000000 | 6445.000000 | 35682.000000 | 6082.000000 | 15.000100 | 500001.000000 |

I hope the count, mean, min and max rows are self explanatory for you. Be aware of the fact, that all null values are ignored here (count of total\_bedrooms is 20,433 not 20,640). The std row shows you the standard deviation, which quantifies the amount of variation between the values. The 25%, 50%, 75% are the percentiles. For an example 75% of the districts have housing\_median\_age of lower than 37 while 50% are lower than 29 and 25% are lower than 18. These are often called the 25th percentile (or 1st quartile), the median and the 75th percentile (or 3rd quartile).

Another good way to explore a given dataset is to plot a histogrom of each numerical attribute. A histogram shows you the number of instances (vertical axes) and the number of the given value range (horizontal axes)

In [9]:

**%matplotlib** inline

housing**.**hist(bins**=**50, figsize**=**(20, 15))

Out[9]:

array([[<matplotlib.axes.\_subplots.AxesSubplot object at 0x114c0dc50>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x1149e5a20>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x114a22898>],

[<matplotlib.axes.\_subplots.AxesSubplot object at 0x114a5d898>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x114a93898>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x114a938d0>],

[<matplotlib.axes.\_subplots.AxesSubplot object at 0x114b01358>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x114b39358>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x110e9f278>]], dtype=object)

What you should notice about the histograms above:

* The attributes have varying scales, which we will discuss later on in this post.
* Many of the histograms are "tail heavy" which means that they extend further to the right of the median than to the left which makes it harder for an algortihm to detect patterns. We will transform the later on.
* The median\_income attribute is not in US Dollar because the data has been scaled and capped at 15.0001 and at 0.4999. This is called a "preprocessed attribute" and is common within machine learning, but you should understand how the data was preprocessed.
* The housing\_median\_age and the median\_house\_value attributes are also capped. That the median\_house\_value is capped could be a serious problem, because this is your label (what you want to predict) and your model could learn that the price never goes beyond that limit. In this case we only have the option to remove the capped one or to collect the right labels for them.

# Train/Test Split

It is important that we now set a part of the data aside. Your brain is an amazing pattern detection system and therefore extremly prune to overfitting. If you would also look at the test set during the visual explaration, which we will do now, you may find some patterns unconsciously within the data, which let you select a particular algorithm that leads to an overfitted model. Because of that you could launch a system that performs not as well on new data than expected.

We use the straight forward "train\_test\_split()" method from sklearn to split our data into train and test subsets. For further information you can look at the [documentation](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html).

In [10]:

train\_set, test\_set **=** train\_test\_split(housing, test\_size**=**0.2, random\_state**=**42)

print(len(train\_set), "Train Instances +", len(test\_set), "Test Instances")

16512 Train Instances + 4128 Test Instances

Now we splitted our data purely random, which is fine if you have a large dataset, but if it is not, you could have a sampling bias. When a company decides to call 10,000 people because of a survey, they want to make sure that these people represent the whole population. For an example the US population consists out of 49% male and 51% female, so a well conducted survey would try to maintain this ratio, which is called **stratified sampling**. The population is divided into homogeneous subgroups, called **strata** and the right number of instances is sampled from each **stratum** to makes sure that the data really represents the population.

Imagine that you talked with real estate experts and they tell you, that the median\_icome attribute is very important, when you want to predict housing prices. If that's the case, you want a test set that is really representative of the income categories of the dataset. Because median\_income is continous, you need to convert it into a categorical attribute. If you look again at the holograms we analyzed detailed above, you may notice that most median income values are clustered around 20,000 - 50,000 but some go far beyond 60,000. It is very important that you don't have too many strata and that each stratum should have a sufficient number of instances. If this is not the case, the estimate of the stratums importance may be biased and your model could think that a stratum is less important.

The code below transforms the median\_income attribute into a categorical one by dividing the median income by 1.5 to limit the number of income categories and rounds it up using "np.ceil()" to have discrete categories. It merges all the categories that are greater than 5 into category 5. The categories are represented in the histogram below the code.

In [11]:

housing["income\_categories"] **=** np**.**ceil(housing["median\_income"] **/** 1.5)

housing["income\_categories"]**.**where(housing["income\_categories"] **<** 5, 5.0, inplace**=True**)

In [12]:

plt**.**hist(housing["income\_categories"])

fig **=** plt**.**gcf()

Lats but not least you need to do stratified sampling based on the income categories. You can use sklearn's "StratifiedShuffleSplit" class:

In [13]:

**from** sklearn.model\_selection **import** StratifiedShuffleSplit

split **=** StratifiedShuffleSplit(n\_splits**=**1, test\_size**=**0.2, random\_state**=**42)

**for** train\_index, test\_index **in** split**.**split(housing, housing["income\_categories"]):

strat\_train\_set **=** housing**.**loc[train\_index]

strat\_test\_set **=** housing**.**loc[test\_index]

The code below just generates the table and compares the different sampling types.

In [14]:

**def** income\_categories\_proportions(data):

**return** data["income\_categories"]**.**value\_counts() **/** len(data)

train\_set, test\_set **=** train\_test\_split(housing, test\_size**=**0.2, random\_state**=**42)

compare\_props **=** pd**.**DataFrame({

"Overall": income\_categories\_proportions(housing),

"Stratified": income\_categories\_proportions(strat\_test\_set),

"Random": income\_categories\_proportions(test\_set),

})**.**sort\_index()

In [15]:

compare\_props

Out[15]:

|  | **Overall** | **Random** | **Stratified** |
| --- | --- | --- | --- |
| **1.0** | 0.039826 | 0.040213 | 0.039729 |
| **2.0** | 0.318847 | 0.324370 | 0.318798 |
| **3.0** | 0.350581 | 0.358527 | 0.350533 |
| **4.0** | 0.176308 | 0.167393 | 0.176357 |
| **5.0** | 0.114438 | 0.109496 | 0.114583 |

Now we want to remove the income\_categoreis attribute because we don't need it anymore.

In [16]:

housing **=** housing**.**drop(['income\_categories'], axis**=**1)

# Extensive Data Exploration

Now it is time for exploring the data. First of all, we want to visualize the geographical data with latitude and longitude. A good way to do this is to create a scatterplot of all the districts. It is important that you set alpha equal to 0.1, because then the scatterplot has a high density and therefore it is much easier to visualize.

In [17]:

housing**.**plot(kind**=**"scatter", x**=**"longitude", y**=**"latitude", alpha**=**0.1)

Out[17]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x114e85e48>

If you are fimiliar with California you can clearly see, the high density in the Bay Area, Los Angelos, San Diego and in the Central Valley around Sacramento and Fresno. Our brains are very good at finding patterns on pictures but you often need to play around with the parameters to make the important patterns really stand out.

Now we will look at the housing prices at the scatterplot below. The color represents the price and the radius of each circle represents the districts population.

In [18]:

housing**.**plot(kind**=**"scatter", x**=**"longitude", y**=**"latitude", alpha**=**0.4,

s**=**housing["population"]**/**100, label**=**"population", figsize**=**(10,7),

c**=**"median\_house\_value", cmap**=**plt**.**get\_cmap("jet"), colorbar**=True**,

sharex**=False**)

plt**.**legend()

Out[18]:

<matplotlib.legend.Legend at 0x1a1f425588>

On the scatterplot we can see, that the housing prices are related to the location (close to the ocean) and to the population density, but we know that the housing prices of costoal districts are not that high in Northern California, so we can't make that rule as simple as that.

### Searching for Correlations:

The housing dataset isn't that large and therefore we can easily compute the correlations between every attribute using the "corr()" method. We will start by looking how much each attribute is correlated to the median house value.

In [19]:

correlations **=** housing**.**corr()

correlations["median\_house\_value"]**.**sort\_values(ascending**=False**)

Out[19]:

median\_house\_value 1.000000

median\_income 0.688075

total\_rooms 0.134153

housing\_median\_age 0.105623

households 0.065843

total\_bedrooms 0.049686

population -0.024650

longitude -0.045967

latitude -0.144160

Name: median\_house\_value, dtype: float64

The coefficient of the correlation ranges from 1 to -1. The closer it is to 1 the more correlated it is and vice versa. Correlations that are close to 0, means that there is no correlation, neither negative or positive. You can see that the median\_income is correlated the most with the median house value. Because of that, we will generate a more detailed scatterplot below:

In [20]:

housing**.**plot(kind**=**"scatter", x**=**"median\_income", y**=**"median\_house\_value", alpha**=**0.1)

Out[20]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x1a1f503b38>

The scatterplot reveals, that the correlation is indeed very strong because we can clearly see an upward trend and the points are not to dispersed. We can also clearly see the price-cap, we talked about earlier, at 500 000 as a horizontal line. Other less obvious lines are around 450 000, 350 000 and 280 000. We may have to remove the corresponding districts to prevent the model from learning to reproduce these data faults.

I hope that the previous explanation and visulizations made you more comfortable with the concepts of exploring data to gain insights. We identified a few very important faults in the data that we need to clean up before we can put the data into the machine learning model, that we will build later on. We also found some interesting correlations between the different attributes and we recognized the tail-heavy distribution, that we will also clean up later on. Of course a lot of things are different at every project you will work on, but the general guidelines will be the same and therefore you now already have a good understanding of some of those guidelines.

Before we now actually prepare the data to fed it into the model, we should think about combinating a few attributes. For example, the number of rooms within a district is of course not very helpful, if you don't know how many households are within that district. You want the number of roms per household. The number of bedrooms isn't that helpful for the same reason, but it would make sense to compare it with the total number of rooms within a household. Also the population per household would be an interesting attribute. I will create these new attributes in the code below and then we will look at the correlation matrix again.

In [21]:

housing["rooms\_per\_household"] **=** housing["total\_rooms"]**/**housing["households"]

housing["bedrooms\_per\_room"] **=** housing["total\_bedrooms"]**/**housing["total\_rooms"]

housing["population\_per\_household"]**=**housing["population"]**/**housing["households"]