MACHINE LEARNING OVERVIEW

January 8, 2020

Nicholas Kim
University College London
MSc Computational Finance

About

This will serve as a short guide to the many functions that can be found in Python's Scikit-Learn, TensorFlow and Keras libraries. I have created this document to hopefully help others learn more about the area of Machine Learning and its capabilities. As a Note this paper, will not serve as a comprehensive guide to Machine Learning but more as a cheat sheet to help one remember certain functions that are contained in the libraries mentioned above.

This guide will continually be updated as libraries are changed and/or new functions are added. I have also not gone through these libraries in full and will add in what I have been able to cover so far.

Please feel free to reach out to me at Nicholaskim46@gmail.com if there are any typos or mistakes throughout this guide.

Contents

1 Training Models/Algorithms

1.1 Linear Regression

Linear Regression simply makes a prediction by computing a weighted sum of the input features, plus a constant term which is called the bias (Intercept)

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

In the equation above:

- $\hat{y}$, is the predicted value

- n, is the number of features

- x_i, is the ith feature values

- θ_j is the jth model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$)

However, this equation can be written much more concisely using a vectorized form:

$$\hat{y} = \mathbf{h}_\theta(\mathbf{x}) = \theta \cdot \mathbf{x}$$

Where we have the following:

- θ, is the model's *parameter vector*, conatining the bias term θ_0 and the feature weights θ_1 to θ_n.

- $\mathbf{x}$, is the instance's *feature vector*, containing x_0 to x_n with x_0 always eqaul to 1.

- h_θ, is the hypothesis function, using the model parameters θ

Now for training the model we want to minimize the Mean Squared Error (MSE). This means that we need to find a value for θ such that it minimizes the MSE. It is also very common to see Root Mean Squared Error (RMSE) as a performance measure of a regression model.

The MSE of a Linear Regression is shown below:

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

1.2 Gradient Descent

Gradient Descent is a optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of this algorithm is to tweak parameters iteratively in order to minimize a cost function.

An intuitive way to think about Gradient Descent is imagine one is trying to find the fastest way to reach a bottom of a valley. A good strategy would be to go downhill in the direction of the steepest slope. This is exactly what Gradient Descent does: it measures the local gradient

of the error function with regard to the parameter vector θ, and it goes in the direction of the descending gradient.

To implement *Gradient Descent*, we need to compute the gradient of the cost function with regard to each model parameter θ_j. Thus, we need to take the partial derivative of the cost function with respect to each θ_j. This can be seen below:

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\theta^\mathsf{T} \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Additionally, instead of computing partial derivatives individually, we can also use the following equation to compute the gradient vector for MSE.

$$\nabla_\theta \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^\mathsf{T} (\mathbf{X}\theta - \mathbf{y})$$

Once we have the gradient vector, which points uphill, we just go in the opposite direction. This means that we need to subtract the $\nabla_\theta \text{MSE}(\theta)$ from θ. Here we will use η as our learning rate parameter for each step. We will multiply the gradient vector by η to determine the size of the downhill step.

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta \text{MSE}(\theta)$$

From the equation above we can see how important η is, when training optimizing training for gradient descent.
Below will be important notes for Gradient Descent:

- θ, this vector is initialized with random values (*Random Initialization*), and it is then improved gradually, taking one step at a time to decrease the cost function.

- **Learning Rate**, η, this parameter determines how large the "learning" will be in each iteration of the algorithm. If the learning rate is too small then it will take too long to converge to the desired state, and on the other hand, if it is too large then the algorithm will never be able to converge as it will be bouncing around the cost function. It is important to note that it is often optimal to set the learning rate dynamically so that initially we have a high learning rate to get faster convergence to the optimal, and then slowly decrease the learning rate as we get closer to the optimal point.

- Finally, not all cost functions are bowl shaped. This means we can have cost functions that have holes, ridges, plateaus, and all other forms of irregular terrain, making convergence of this alorgithm difficult. However, this can be helped by the optimization of the learning rate.

1.3 Polynomial Regression

Polynomial Regression, can be done by adding powers to each feature, then training a linear on this set of extended features. Using Scikit-Learn's `PolynomialFeatures` class, we can transform our training data using the following code:

```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

m = 100
X = 6 * np.random.rand(m, 1)
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)

poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)

lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
```

Now `X` is transformed into `X_poly` where it now contains the original features of $\mathbf{X}$ plus the squares of this feature (Also note that as we increase variables, this fuction will also include all the cross terms. For example, if we have variables a and b with degree=3, we add the features $a^2, a^3, b^2, b^3, ab, a^2b, ab^2$). We then fit the new features `X_poly` with Scikit-Learns's `LinearRegression`, which returns us a quadratic polynomial regression.

Warning: *PolynomialFeatures(degree=d)* transforms an array continaing n features into an array containing $\frac{(n+d)!}{d!n!}$ features. So we must take care in the explosion of the number of features.

Bias/Variance Trade Off

- **Bias**

    - This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.

- **Variance**

    - This part is due to the models excessive sensitivity to small variations training data. A model with many degrees of freedom (such as high-degree) polynomial model) is likely to have high variance and thus overfit the training data.

- **Irreducible Error**

- This is due to the noisiness of data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers)

- Generally, increasing a model complexity till typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is the reason for this so-called trade off

- The optimal strategy would be to find the intersection that minimizes both the bias and variance.

1.4 Regularized Linear Models

Regularizing linear models is a great way to reduce the chance of overfitting the data that it is using to be trained. This is the case as it helps reduce the amount of degrees of freedom that the model has.

In this section we will take a look at three different regularized linear models: Ridge Regression, LASSO, Elastic Net.

1.4.1 Ridge Regression

Ridge Regression, is a regualrized linear regression with a regularization term eqaul to: $\alpha \sum_{i=1}^{n} \theta_i^2$, which is then added to the cost function. This forces the algorithm to not only fit the data but to keep the models weights as small as possible.

It is important to note that we want to keep the regularization term during **training only** and once deployed, the regularization term should be dropped.

The hyperparameter α controls how much we want to regularize the model. If $\alpha = 0$ then the model is just a linear regression, however, if α is very large, then all the weights end up very close to zero. Where the result will be a flat line going through the data's mean.

Below will be the total cost function for Ridge Regression:

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Note that the biased term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define $\mathbf{w}$ as the vector of feature weights (θ_1 to θ_n), then the regularization term is equal to $\frac{1}{2}(\|\mathbf{w}\|_2)^2$ represents the l_2 norm of the weight vector.

Intuitive Explanation of l_2 norm

The l_2 norm penalizes the square of the weights in the model. Therefore, it is much more inclined to push down the big weights as compared to the smaller weights. This is the reason

why we see weights get pushed down close to zero for this model but never all the way to zero.

Remember, it is important to scale the data (e.g., using *StandardScaler* in Pythons Scikit-Learn) before performing Ridge Regression, as the model is sensitive to the scale of input features. This holds for most regularized models.

1.4.2 LASSO

Lease Absolute Shrinkage and Selection Operator Regression (LASSO) is another version of a regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, but it uses a l_1 norm of the weight vector instead of half the square of the l_2 norm.

Below will be the LASSO Regression Cost Function:

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

A very important characteristic of Lasso Regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero). In other words, Lasso Regression automatically performs feature selection and outputs a *sparse model* (i.e., with few nonzero weights).

Intuitive Explanation of l_1 norm

LASSO tends to send coefficients to zero because it tries to minimize the absolute value of the weights, thus it is inclined to make big weights smaller and small weights smaller as well. Therefore, it tends to drive the weights to 0, which then brings upon a sparse weight vector.

1.4.3 Elastic Net

Elastic Net is the middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and this can be controlled with the mix ratio r. When $r = 0$, Elastic Net is equal to Ridge Regression, and when $r = 1$, it is equivalent to Lasso Regression.

Below will be the cost function for Elastic Net:

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

1.4.4 When to use Linear Regression, LASSO, Ridge, or Elastic Net?

It is almost always perferable to have at least a little bit of regularization, so generally, plain Linear Regression should be avoided. Ridge Regression, is usually a good default for solving linear problems, however, if we suspect that there may be only a few features that are useful, we should prefer to use Lasso or Elastic Net. As discussed earlier, it is preferred to use Lasso

or Elastic Net when we want to try and reduce the least important features in a dataset to zero.

In general, Elastic Net is preferred over Lasso as sometimes the model can behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

1.5 Logistic Regression

Logistic Regression (Also called *Logit Regression*) is a regression that is commonly used to estimate the probability that an instance belongs to a particular class. If the probability is greater than 50%, then the model predicts that the instance belongs to that class and otherwise it does not. This trait makes it a *binary classifier*.

1.5.1 Estimating Probabilities

Logistic Regression works just like a Linear Regression model, it computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model, it outputs the *logistic* of this result (See Below).

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma(\mathbf{x}^\mathsf{T}\theta)$$

The logistic – noted $\sigma(\cdot)$ – is a *sigmoid function* (i.e., S-Shaped) that outputs a number between 0 and 1. It is defined below:

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

Once the Logistic Regression model is able to estimate the probability $\hat{p} = h_\theta(\mathbf{x})$ that an instance $\mathbf{x}$ belongs to the positive class, it can make the prediction for $\hat{y}$ easily using the function below:

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

Note: The score t is often called the *logit*. This comes from the fact that the logit function is defined as $\text{logit}(p) = \log(\frac{p}{1-p})$, this is the inverse of the logistic function. The logit is also called the *log-odds*, since it is the log of the ratio between the estimated probability for the positive class and the estimated probability for the negative class.

1.5.2 Training and Cost Function

The objective of training a Logistic Regression is to set the parameter vector θ so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$). This idea is captured by the cost function shown below for a single training instance $\mathbf{x}$:

$$c(\theta) = \begin{cases} \text{-log}(\hat{p}) & \text{if } y = 1 \\ \text{-log}(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

Intuitively, we can see the cost function makes sense. As t approaches 0 we can see that $-\log(t)$ grows very large, so the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance. On the other hand, we can see that the cost will be very small is the model is able to estimate the instance correctly.

The cost function over the whole training set is the average cost over all training instances. It can be shown in a single expression called the *log loss*, shown below:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})]$$

There is no closed-form equation to compute the value of θ that minimizes this cost function. However, this cost function is convex, so Gradient Descent (or other optimization algorithms) is guaranteed to find the global minimum.

The partial derivatives of the cost function w.r.t. the jth model parameter θ_j is shown below:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (\sigma(\theta^{\mathsf{T}} \mathbf{x}^{(i)}) - y^{(i)}) \mathbf{x}_j^{(i)}$$

For each instance, it computes the prediction error and multiplies it by the jth feature value, and then it computes the average over all training instances. Once we are able to get the gradient vector containing all the partial derivatives, we can use it in Batch Gradient Descent algorithm (So on with Stochastic Gradient Descent and Mini-batch Gradient Descent).

2 Classification

In this section, we will be covering the basics of classification machine learning techniques. Topics will range from what data to use for learning, how to train a Multiclass/binary classifiers, look over different performance measures and finally looking over multioutput classifiers.

2.1 MNIST

In this section we will be utilizing the MNIST dataset, which is made up of 70,000 images of digits written by hand. Each image is labeled with the digit it is an image of. This is one of the most popular datasets for intially learning about the world of Machine Learning (Equivalent to the "Hello World" program when first learning how to code).

Fortunately, Scikit-Learn provides a very easy way to obtain this data. Using the following block of code we can obtain this dataset:

```python
from sklearn.datasets import fetch_openml
mnist = fetch_openml("mnist_784", version=1)
```

```
mnist.keys()

# Below will be the dictionary for this dataset
'''
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details',
           'categories', 'url'])
'''
```

Datasets from Scikit-Learn have mostly the same dictionary structure:

- A `DESCR` key describes the dataset

- A `data` key contains an array with one row per instance and one column per feature

- A `target` key containing an array with all the labels for each respective image that is in the data

Lets take a look at the shape of the dataset:

```
X, y = mnist["data"], mnist["target"]
X.shape # (70000, 784)
y.shape # (70000,)
```

Here we see, as mentioned earlier, that there is 70,000 images with each image containing 784 features. There are 784 features as each image is 28 x 28 pixels, and each feature represents the pixels density ranging from 0 (white) to 255 (black). Additionally, we see that the target data is just 70,000 labels for each respective image in X.

Finally, we should always split the data before inspecting the data more closely. We can make the test and training sets in the following manner:

```
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]
```

Note: that this dataset has already been shuffled. However, if this dataset has not been shuffled then it is crucial that we do this before splitting the data as we do not want some folds when doing cross-validation to be missing digits.

2.2   Training a Binary Classifier

For simplification reasons, we will first only try to identify one digit, number 5. This classifier will be a *binary classifier*, capable of distinguishing between just 2 classes 5 and not a 5.

First we will create target vectors for this classification task:

```
y_train_5 = (y_train == 5) # True for all 5s, False for all others
y_test_5 = (y_test == 5)
```

For this example we will be using the *Stochastic Gradient Descent* (SGD) classifier, using Scikit-Learn's `SGDClassifier` class. This classifier is advantageous because it is able to handle large datasets efficiently. This is the calse because SGD deals with training instances independently, one at a time.

Below will be some code that is implementing this classifier:

```
from sklern.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train_5, y_test_5)
```

Now that the model is trained we, hope, that we can now predict with reasonable accuracy that a number is a 5 and not a 5 if it is some other digit.

2.3 Performance Measures

Here we find that evaluating a classifier tends to be much trickier than evaluating a regression based model. In this section, we will be spending a lot of time going over the many different performance measures there are available.

2.3.1 Measuring Accuracy Using Cross-Validation

A common way to evaluate a model is to just use cross-validation. Here we will be using a off the shelf cross validation function, but if you need more control over the cross-validation processs then you can always build your own.

Below will be the implementation using Sciki-Learn's `cross_val_score()` function to evaluate our `SGDClassifier` we trained earlier. Note, that we are using K-folds cross-validation when using the python function `cross_val_score()`.

```
from sklearn.model_selection import cross_val_score
# Splits training set into 3 folds, then making predictions and
# evaluating them on each fold using a model trained on the remaining
# folds.
cross_val_score(sgd_clf, X_train, y_train, cv = 3, scoring='accuracy')

# Output:
# array([0.96355, 0.93795, 0.95615])
```

After cross-validation we see that the model has an accuracy of above 93% on all cross-validation folds. However, there is a catch to this. We will now fit a "very dumb" classifier that just classifies every image as a "not-5" class:

```python
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(x), 1), dtype=bool)

never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train, cv=3, scoring='accuracy')

# Output:
# array([0.91125, 0.90855, 0.90915])
```

Woah, we have over 90% accuracy on the dataset by just classifying everything as not a 5. This is because we only have about 10% of the images as 5s, so if we always guess that the image is not a 5 then we will be right about 90% of the time.

The above is a prime example why accuracy is generally not the preferred performance measure for classifiers, especially when we are dealing with *skewed datasets* (i.e., when some classes are much more frequent then others).

2.3.2   Confusion Matrix

Another way to evaluate performance of a classfier is to look at the *Confusion Matrix*. In general, the idea of this matrix is to count the number of times instances of a class A are classified as a class B.

To create the confusion matrix, we need to first have a set of predictions so that they can be compared to the actual targets. As we generally do not want to touch the test set until the model is complete, we can use `cross_val_predict()` instead to generate our predictions against the target:

```python
from sklearn.model_selection import cross_val_predict()
from sklearn.metrics import confusion_matrix

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

confusion_matrix(y_train_5, y_train_pred)

'''
```

```
Output:
Confusion Matrix Below:
array([[53057,  1522],
       [ 1325,  4096]])
'''
```

Similarly as above, `cross_val_predict()`, also performs K-fold cross-validation, but instead of returning scores, it returns the predictions made on each test fold. Now for the confusion matrix:

Confusion Matrix Explanation

- Each **Row** in a confusion matrix represents an *actual class*

- Each **Column** in a confusion matrix represents a *predicted class*

In the case for this model, the **first row** of the matrix above considers non-5 images (negative class): 53,057 of them were correctly classified as non-5s (called *True Negatives*), while the remaining 1,522 were wrongly classified as 5s (*False Positives*). The **second row**, considers the images of 5s (*positive class*): 1,325 were wrongly classified as non-5s (*false negatives*), while the remaining 4,096 were correctly classified as 5s (*True Positives*). Additionally, a perfect classifier would only have non-zero values on the main diagonal.

2.3.3   Precision and Recall

Although the *confusion matrix* may give a lot of information, sometimes it is better to look at a more straight-forward metric. Both *Precision* and *Recall* are great for this task.

Precision

Precision, can be defined as the accuracy of the positive prediction for the classifier we are evaluating. Below will be the formula to calculate precision:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where:

- TP $\rightarrow$ The number of true positives

- FP $\rightarrow$ The number of false positives

Recall

Recall, can be defined as the *sensitivity* or the *true positive rate* (TPR): this is the ratio of positive instances that are correctly detected by the classifier. The equation for Recall can be defined as follows:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where, FN, is the number of false negatives. Additionally, below we can see a more detailed version of the confusion matrix to help clear any confusion:
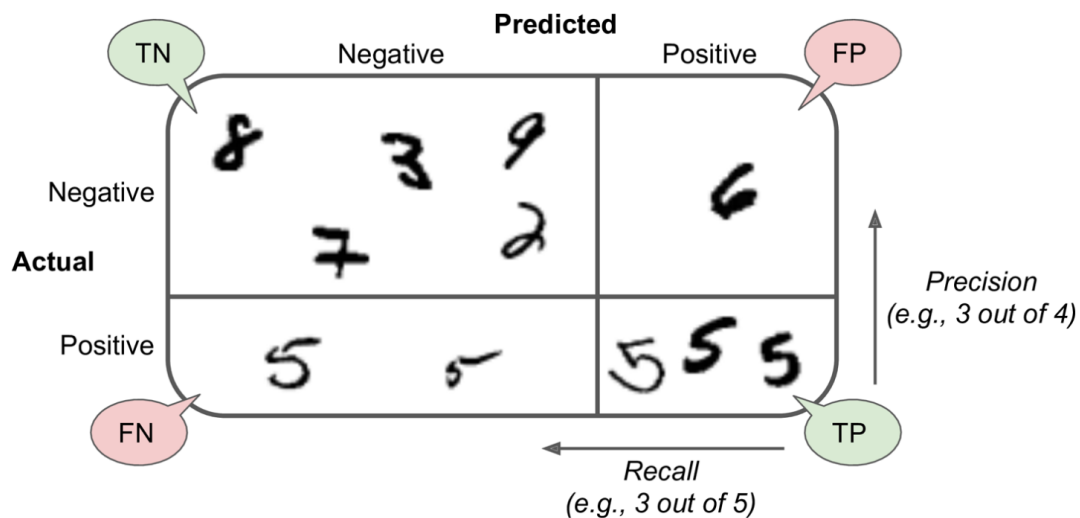
Precision and Recall

Scikit-Learn provides several functions to help compute classifier metrics, including precision and recall. See below an example:

```
from sklearn.metrics import precision_score, recall_score

precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
# Output: 0.7291

recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
# Output: 0.7556
```

Now we can see that the classifier is definitely not as good as we originally thought. From **Precision**, we now see that when the classifier claims an image is of a 5, it is only correct about 72.9% of the time. Additionally, from **Recall**, we can see that the classifier only detects 75.6% of the 5s.

In addition, we often find it useful to combine both precision and recall into a single metric which is known as the **F_1 score**. This new metric can be defined as follows:

$$\mathbf{F_1} = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{\text{TP}}{\text{TP} + \frac{\text{FN}+\text{FP}}{2}}$$

As follows, Scikit-Learn has a built in function for this as well. We can call `f1_score()` function.

```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
# Output: 0.7421
```

Note: that the F_1 score favors classifiers that have similar precision and recall. This may not always be that case that we want. Some cases may call for mostly caring about precision, and on other cases it may be best to only care a lot about recall.

We can also not have both high precision and recall, as if we increase one the other tends to decrease. This is known as the *Precision/Recall Trade-Off*.

2.3.4 ROC Curve

The *Revceiver Operating Characteristic* (ROC) curve is another common way to evaluate a binary classifiers ability to predict correct classes. The ROC curve is created by plotting the *True Positive Rate* (Also known as Recall) against the *False Positive Rate* (FPR). The FPR is the ratio of negative instances that are incorrectly classified as positive. It is also equal to $1-$ the *True Negative Rate* (TNR), which is the ratio of negative instances that are correctly classified as negative. TNR is also called *Specificity*.

Note: As a general rule, it is better to use the *Precision/Recall Curve* when the positive class is rare or when we care more about the false positive than the false negatives. Otherwise, it is better to use the ROC curve.

2.4 Multiclass Classification

In this section we will be discussing how we can train classifiers that are choosing between more than just two separate classes. Some algorithms (SGD classifiers, Random Forest Classifiers, and Naive Bayes Classifiers) are capable of handling multiple classes natively. However, others (Logisitc Regression, SVM) are strictly binary classifiers, but as we will see, there are strategies so that we can perform multiclass classification with binary classifiers.

There are 2 strategies for making it possible for binary classifiers to have the ability to classify multiple classes. The strategies are known as **One-Versus-Rest** (OvR) and **One-Versus-One** (OvO).

One-Versus-Rest

This strategy is fairly basic in that, for example, if we want to train binary classifiers to classify digit images into 10 classes (0 to 9), we can train 10 binary classifiers, one for each digit (0-detector, 1-detector, etc.). Then when we want to classify an image, we get a decision score from each classifier for that image and we select the class whose classifier outputs the highest score.

One-Versus-One

The second strategy is to train binary classifiers for every pair of digits to distinguish 0s and 1s, 0s and 2s, 1s and 2s, and so on. If there are N classes then we will need to train $\frac{N(N-1)}{2}$ classifiers. For the MNIST data we will need to train 45 binary classifiers. When we want to classify an image, we have to run the image through all 45 classifiers and see which class wins the most duels. The main advantage of this strategy is that each classifier only needs to be trained on a part of the training set for the two classes it must distinguish.

However, some algorithms, such as SVM, scale very poorly with size of the training set. Therefore, it is preferred that these algorithm use the OvO strategy as it is faster to train several classifiers on small training sets than a few on large training sets (Although in most cases for binary classifiers, OvR is preferred).

3 Support Vector Machines

Support Vector Machines (SVM), are one of the most popular models in Machine Learning. SVMs are capable of performing linear or nonlinear classification, regression and outlier detection. In general, SVMs are particularly strong suited in classification tasks that include complex datasets that are small or medium sized, as they do not scale well into larger datasets.

3.1 How SVMs Work

In this section we will be covering the mechanics behind how SVMs make predictions and how their training algorithms work. A few words on notations, the bias terms will be called b, and the feature weights vector will be called $\mathbf{w}$. Additionally, no bias feature will be added to the input feature vectors.

Decision Function and Predictions

A linear SVM classifier model predicts the class of a new instance $\mathbf{x}$ by simply computing the decision function: $\mathbf{w}^\intercal\mathbf{x} + b = w_1x_1 + \cdots + w_nx_n + b$. If the result is positive, the predicted class $\hat{y}$ is the positive class (1), and otherwisw it is the negative class (0). Below will be the function for a Linear SVM classifer's predictions:

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^\intercal\mathbf{x} + b < 0 \\ 1 & \text{if } \mathbf{w}^\intercal\mathbf{x} + b \geq 0 \end{cases}$$

The figure below will show the plot of the decision function. It is a 2D plane because the dataset we are training the model on has only 2 features: petal width and petal length (Iris Datset). The decision boundary is the set of points where the decision function is equal to 0: it is the intersection of 2 planes, which is a straight line (represented by the thick solid line).
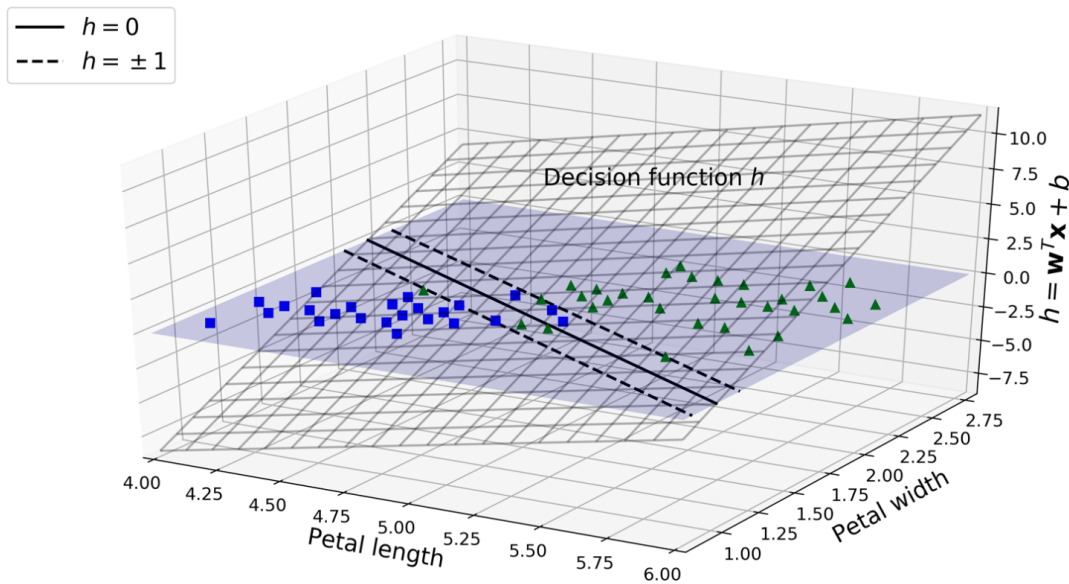
The dashed lines represent the points where the decision function is equal to 1 or -1. **Note:** they are parallel and at equal distance to the decision boundary, and they form a margin around it. When we train a linear SVM classifier, we are finding the values of $\mathbf{w}$ and b that make this margin as wide as possible while avoiding margin violations (*hard margin*) or limiting them (*soft margin*).

Training Objective

Consider the norm of the weight vector, $\|\mathbf{w}\|$, we find that it is equivalent to the slope of the decision function. If we were to divide this slope by 2, we will also find that the points where the decision function is equal to ± 1 are going to twice as far away from the decision boundary. This can be seen in the figure below:
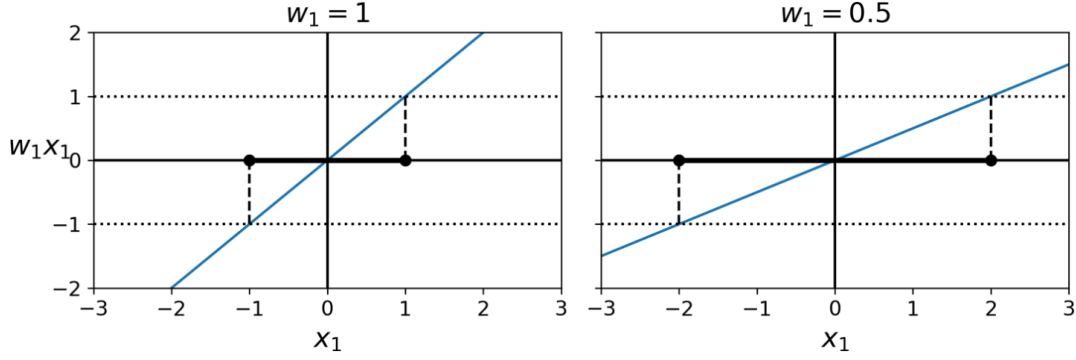
Therefore, we want to minimize $\|\mathbf{w}\|$ to get the largest possible margin. Additionally, if we want to avoid any margin violations (hard margin), then we need the decision function to be greater than 1 for all positive training instances and lower than -1 for negative training instances.

Further, if we define $t^{(i)} = -1$ for negative instance (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express the constraint as $t^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)}+b) \geq 1$ for all instances.

Finally, we can express the **hard margin** linear SVM classifier objective as the constrained optimization problem below:

$$\min_{\mathbf{w},b} \quad \frac{1}{2}\mathbf{w}^\mathsf{T}\mathbf{w}$$
$$\text{subject to:} \quad t^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

Note: We are minimizing $\frac{1}{2}\mathbf{w}^\mathsf{T}\mathbf{w}$, which is just equal to $\frac{1}{2}\|\mathbf{w}\|^2$, rather than minimizing $\|\mathbf{w}\|$. This is because $\frac{1}{2}\|\mathbf{w}\|^2$ has a nice derivative, which is just $\mathbf{w}$, whereas $\|\mathbf{w}\|$ is not differentiable at $\mathbf{w} = 0$. In general optimization algorithms work much better on differentiable functions.

For the **soft margin** objective, we need to a *slack variable* $\zeta^{(i)} \geq 0$ for each instance. $\zeta^{(i)}$ can be described as how much the ith instance is allowed to violate the margin (decision boundary). However, now we have two conflicting objectives:

- First, is to minimize the slack variables as much as possible to reduce the margin violations

- Second, make $\frac{1}{2}\mathbf{w}^\mathsf{T}\mathbf{w}$ as small as possible to increase the margin

Finally, this is where the C hyperparameter comes in. This hyperparameter allows us to define the tradeoff between these two objectives, thus, giving us the constrained optimization problem for the **soft margin** linear SVM classifier:

$$\min_{\mathbf{w},b,\zeta} \quad \frac{1}{2}\mathbf{w}^\mathsf{T}\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}$$
$$\text{subject to:} \quad t^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{for } i = 1, 2, \cdots, m$$
$$\zeta^{(i)} \geq 0$$

19

Quadratic Programming

Hard and Soft Margin problems are both convex quadratic optimization problems with linear constraints. This is commonly known as *Quadratic Programming* (QP) problems. There are many algorithms that can be used for this task. Take a look at this book for more detailed explanations of these algorithms.

The general Quadratic Programming problem to solve is given by:

$$\min_{\mathbf{p}} \quad \frac{1}{2}\mathbf{p}^\mathsf{T}\mathbf{H}\mathbf{p} + \mathbf{f}^\mathsf{T}\mathbf{p}$$

subject to: $\mathbf{Ap} \le \mathbf{b}$

where $\begin{cases} \mathbf{p} & \text{is an } n_p - \text{dimensional vector } (n_p = \text{ number of parameters}) \\ \mathbf{H} & \text{is an } n_p \times n_p \text{ matrix} \\ \mathbf{f} & \text{is an } n_p - \text{dimensional vector} \\ \mathbf{A} & \text{is an } n_c \times n_p \text{ matrix } (n_c = \text{ number of constraints}) \\ \mathbf{b} & \text{is an } n_c - \text{dimensional vector} \end{cases}$

Note that the expression $\mathbf{Ap} \le \mathbf{b}$ defines a n_c constraints: $\mathbf{p}^\mathsf{T}\mathbf{a}^{(i)} \le b^{(i)}$ for $i = 1, 2, \cdots, n_c$, where $\mathbf{a}^{(i)}$ is the vector containing the elements of the ith row of $\mathbf{A}$ and $b^{(i)}$ is the ith element of $\mathbf{b}$.

As an example, we can verify the QP parameters for a hard margin linear SVM classifier below:

- $n_p = n + 1$, where n is the number of features (+1 is for the bias term)

- $n_c = m$, where m is the number of training instances

- $\mathbf{H}$ is the $n_p \times n_p$ identity matrix, except with a zero in the top-left cell (to ignore the bias term)

- $\mathbf{f} = 0$, an n_p−dimensional vector full of 0s

- $\mathbf{b} = -1$, an n_c−dimensional vector full of -1s

- $\mathbf{a}^{(i)} = -t^{(i)}\dot{\mathbf{x}}^{(i)}$, where $\dot{\mathbf{x}}^{(i)}$ is equal to $\mathbf{x}^{(i)}$ with an extra bias feature $\dot{\mathbf{x}}_0 = 1$

Thus, 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 $\mathbf{p}$ will contain the bias term $b = p_0$ and the feature weights $w_i = p_i$ for $i = 1, 2, \cdots, n$. Similarily, we can solve the soft margin problem using a QP solver as well.

The Dual Problem

For more indepth explanation of this problem reference this site.

In its most basic form the *Dual Problem (Duality)* is when we have a given constrained optimization problem, known as the *primal problem*, and it is possible to express a different but closely related problem, called the *dual problem*. Typically, the dual problem gives a lower bound to the solution of the primal problem.

Fortunately, the SVM constrained optimization problem meets these conditions, so we have the choice to solve the primal or dual problem. The optimal solutions for both will be the same. To derive the unconstrained optimization problem for the hard margin SVM, we need to use *Lagrange Multipliers*.

Given the constraints we shown earlier, we can write down the *Generalized Lagrangian* for the hard margin problem for linear SVMs. $\alpha^{(i)}$, is the variable that is called the *Karush-Kuhn-Tucker* (KKT) multipliers. Below will be the Lagrangian:

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\mathbf{w}^{\mathsf{T}}\mathbf{w} - \sum_{i=1}^{m}\alpha^{(i)}(t^{(i)}(\mathbf{w}^{\mathsf{T}}\mathbf{x}^{(i)} + b) - 1)$$

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

As follows with Lagrange multipliers, we can compute the partial derivatives and locate the optimal points. If there is a solution, it will be among the stationary points $(\hat{\mathbf{w}}^{\mathsf{T}}, \hat{b}, \hat{\alpha})$ that respect the KKT Conditions:

- Respect the problems constraints: $t^{(i)}(\mathbf{w}^{\mathsf{T}}\mathbf{x}^{(i)} + b) \geq 1$ for $i = 1, 2, \cdots, m$

- Verify $\hat{\alpha}^{(i)} \geq 0$ for $t = 1, 2, \cdots, m$

- Either $\hat{\alpha}^{(i)} = 0$ or the ith constraint must be an *active constraint*, meaning it must hold by equality: $t^{(i)}(\mathbf{w}^{\mathsf{T}}\mathbf{x}^{(i)} + b) = 1$. This condition is called the *complementrary slackness* condition. It implied that either $\hat{\alpha}^{(i)} = 0$ or the ith instance lies on the boundary (it is a support vector).

Note that the KKT conditions are necessary conditions for a stationary point to be a solution of the constrained optimization problem. Fortunately, the SVM optimization problem meets all these conditions, so any stationary point that meets the KKT conditions is guaranteed to be a solution to the constrained optimization problem.

We can compute the partial derivatives of the generalized Lagrangian with regard to $\mathbf{w}$ and b:

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^{m}\alpha^{(i)}t^{(i)}\mathbf{x}^{(i)}$$

$$\frac{\partial}{\partial b}\mathcal{L}(\mathbf{w}, b, \alpha) = -\sum_{i=1}^{m}\alpha^{(i)}t^{(i)}$$

We then set these partial derivatives equal to zero, and we can get the following:

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

$$\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} = 0$$

Now we can plug these results into the definition of the generalized Lagrangian and we get the following dual problem form for the SVM problem:

$$\mathcal{L}(\hat{\mathbf{w}}, \hat{b}, \alpha) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\mathsf{T}} \mathbf{x}^{(j)} - \sum_{i=1}^{m} \alpha^{(i)}$$

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

We now want to find the vector $\hat{\alpha}$, that minimizes this function, while $\hat{\alpha}^{(i)} \geq 0$ for all instances.

Once we find the optimal $\hat{\alpha}$, we can compute $\hat{\mathbf{w}}$ using the partial derivative w.r.t. $\mathbf{w}$ of the Lagrangian. To compute $\hat{b}$, we can use the fact that a support vector must satisfy $t^{(i)}(\hat{\mathbf{w}}^{\mathsf{T}} \mathbf{x}^{(i)} + \hat{b}) = 1$, so if the kth instance is a support vector (i.e., $\hat{\alpha}^{(k)} > 0$), we can use it to compute $\hat{b} = t^{(k)} - \hat{\mathbf{w}}^{\mathsf{T}} \mathbf{x}^{(k)}$. However, it is often preferred to compute the average over all support vectors to get a more stable and precise value as show below: (Bias term estimation using the dual form)

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

Altogether it is found that the dual problem is faster to solve than the primal one when the number of training instances is smaller than the number of features. However, more importantly the dual problem allows us to use something called the *kernel trick* prossible, while the primal does not (More explained in next section).

Kernelized SVMs

Suppose we want to apply a second-degree polynomial transformation to a two-dimensional training set, then train a linear SVM classifier on the transformed training set. Below we can see the second-degree polynomial mapping function ϕ we want to apply:

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1 x_2 \\ x_2^2 \end{pmatrix}$$

Notice: that the transformed vector is 3D instead of 2D. Now we take a look at how 2D vectors, **a** and **b**, if we apply this second-degree polynomial mapping and then compute the dot product of the transformed vectors we get the following:

$$\phi(\mathbf{a})^\mathsf{T}\phi(\mathbf{b}) = \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^\mathsf{T} \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2$$

$$= (a_1b_1 + a_2b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^\mathsf{T} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^\mathsf{T}\mathbf{b})^2$$

From the above, we can see that the dot product of the transformed vectors is equal to the sum of the square of the dot product of the original vectors $\phi(\mathbf{a})^\mathsf{T}\phi(\mathbf{b}) = (\mathbf{a}^\mathsf{T}\mathbf{b})^2$.

Key Insight: if we want to apply the transformation ϕ to all training instances, then the dual problem (mentioned earlier), will be the dot product $\phi(\mathbf{x}^{(i)})^\mathsf{T}\phi(\mathbf{x}^{(j)})$. But if ϕ is the second-degree polynomial transformation defined earlier, then we can replace this dot product of the transformed vectors simply by $(\mathbf{x}^{(i)\mathsf{T}}\mathbf{x}^{(j)})^2$. Therefore, we do not need to transform the training instances at all; just replace the dot product by its square in the dual problem for SVM defined earlier. The following result will be the same as if we went through the trouble of transforming the training set then fitting a linear SVM algorithm, however, this "trick" makes the process much more computationally efficient.

In Machine Learning, a *kernel* is a function capable of computing the dot product $\phi(\mathbf{a}^\mathsf{T})\phi\mathbf{b}$, based only on the original vectors **a** and **b**, without having to compute (or even know) the transformation ϕ. Below will be a list of some of the most commonly used kernels:

$$
\begin{aligned}
\text{Linear:} \quad & K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\mathsf{T}\mathbf{b} \\
\text{Polynomial:} \quad & K(\mathbf{a}, \mathbf{b}) = (\gamma\mathbf{a}^\mathsf{T}\mathbf{b} + r)^d \\
\text{Gaussian RBF:} \quad & K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma\|\mathbf{a} - \mathbf{b}\|^2) \\
\text{Sigmoid:} \quad & K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma\mathbf{a}^\mathsf{T}\mathbf{b} + r)
\end{aligned}
$$

Mercer's Theorem: According to this theorem if a function $K(\mathbf{a}, \mathbf{b})$ respects a few mathematical conditions called *Mercer's Conditions* (e.g., K must be continous and symmertric in its arguements so that $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), then there exists a function ϕ that maps **a** and **b** into another space (possible with much higher dimensions) such that $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^\mathsf{T}\phi(\mathbf{b})$. We can use K as a kernel because we know that ϕ exists, even if we dont know exactly what ϕ is. For example int he case of the Gaussian RBF kernel, it can be shown that ϕ maps each training instance to an infinite-dimensional space.

Note, that some frequently used kernels (such as the sigmoid kernel) do not respect all of the Mercer's conditions, yet they generally work well in practice.

Earlier in this section we covered how to get to the dual solution to the primal solution in the case of the linear SVM classifier. But if we were to apply the kernel trick, we will end up

with equations that include $\phi(x^{(i)})$, which may be large or even infinite, so we are not able to compute it. Then we ask the question how can we make predictions without knowing $\hat{\mathbf{w}}$? Well we can plug in $\hat{\mathbf{w}}$ from the partial derivatives we took earlier, into the decision function for a new instance $\mathbf{x}^{(n)}$, and we get an equation with only dot products between input vectors. This is what makes it possible to use the kernel trick. Below will be how we make predictions with a kernelized SVM:

$$
\begin{aligned}
h_{\hat{\mathbf{w}},\hat{b}}\left(\phi(\mathbf{x}^n)\right) &= \hat{\mathbf{w}}^\intercal \phi\left(\mathbf{x}^{(n)}\right) + \hat{b} = \left(\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \phi\left(\mathbf{x}^{(i)}\right)\right)^\intercal \phi\left(\mathbf{x}^{(n)}\right) + \hat{b} \\
&= \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \left(\phi\left(\mathbf{x}^{(i)}\right)^\intercal \phi\left(\mathbf{x}^{(n)}\right)\right) + \hat{b} \\
&= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \hat{\alpha}^{(i)} t^{(i)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}\right) + \hat{b}
\end{aligned}
$$

Note that since $\alpha^{(i)} \neq 0$ only for support vectors, making predictions involves computing the dot product of the new input vector $\mathbf{x}^{(n)}$ with only the support vectors, not all the training instances. Additionally, we will use the same trick to compute the bias term $\hat{b}$:

$$
\begin{aligned}
\hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \left(t^{(i)} - \hat{\mathbf{w}}^\intercal \phi\left(\mathbf{x}^{(i)}\right)\right) \\
&= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \left(t^{(i)} - \left(\sum_{j=1}^{m} \hat{\alpha}^{(j)} t^{(j)} \phi\left(\mathbf{x}^{(j)}\right)\right)^\intercal \phi\left(\mathbf{x}^{(i)}\right)\right) \\
&= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \left(t^{(i)} - \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)}}}^{m} \hat{\alpha}^{(j)} t^{(j)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)\right)
\end{aligned}
$$

Finally, we have finished understanding the inner workings of the kernelized SVM.

3.2 Linear SVM Classification

The fundamental idea behind *Support Vector Machines* is to find a line that can separate two separate classes (Data would be known as *linearly separable*). The goal of a SVM classifier is to find a way to separate the two classes, but to simultaneously also fit the widest possible street for separating the two classes. This is called *Large Margin Classification*.

Note: adding more training instaces that are "off the street" will not affect the decision boundary at all. The decision boundary is determined (or "supported) by the instances located on the edge of the street only. These instances are called the *support vectors*.

Important, SVMs are sensitive to feature scaling, so we must be wary when we are training models on unscaled data or when we are doing any feature engineering.

3.2.1   Hard Margin Classification

Hard Margin Classification is when we are looking to very strict with the classification of our instances. We can impose a rule there all instances must be off the decision boundary and on the right side. This is the basic rule of hard margin classification.

However there are two main issues that are associated with this type of classification:

- This type of classification **only** works if the data is linearly separable.

- Training a model like this is also very sensitive to outliers. For example, with just one outlier on the wrong side of the class will make it impossible for there to be a hard margin. This will also likely lead to having the model generalize very poorly.

Therefore, in most cases it is not preferred to use hard margin classification when training a SVM as it is not able to handle many real world datasets.

3.2.2   Soft Margin Classification

On the other hand, *Soft Margin Classification* is a much more flexible classification system. Here the objective is to find the right balance between keeping the decision boundary as large as possible and limiting the *margin violations* (i.e., instances that end up in the middle of the boundary or on the wrong side).

When creating a SVM model using Scikit-Learn, there is a hyperparameter `C` that allows us to regularize the model. If we set a low value for `C`, then we end up with a more flexible model, whereas, if we set a high `C`, then we get a model that much more similar to *hard margin classification*.

Example of SVM in Python

The following code will be using Scikit-Learn to train a linear SVM model on the iris dataset. The code will scale the features and then train the model. The objective of this model it to detect *Iris Virginica* flowers in the iris dataset.

```python
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import Linear SVC

iris = datasets.load_iris() # Iris Dataset
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)

svm_clf.predict([[5.5, 1.7]])
# Output: array([1.])
```

Additionally, instead of using the `LinearSVC` class, we can also use the `SVC` class with a linear kernel. When creating the SVC model, we will just need to specify `SVC(kernel="linear", C=1)`.

3.3 Non-Linear SVM Classification

There are many times when datasets are not anywhere close to being linearly separable. In this section we will be covering different ways to go about creating nonlinear SVM classifiers. The first approach is to add more features, such as polynomial features, which in some cases may lead to the dataset being linearly separable. As an example below we have 2 graphs, on the left there is just one variable, x_1, and we can clearly see that the data is not linearly separable. Now on the right graph, we add a second feature $x_2 = (x_1)^2$, resulting in a 2D dataset that is perfectly linearly separable.
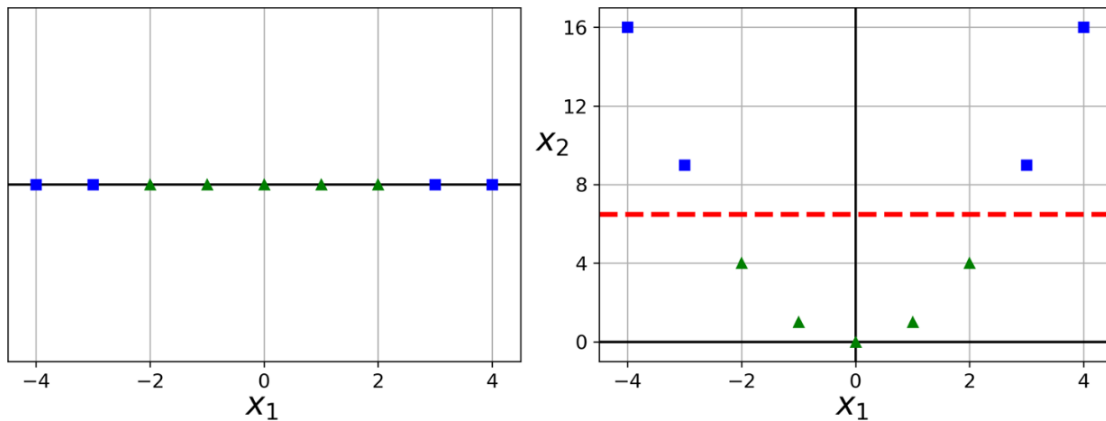
To implement this using Scikit-Learn, we can create a `Pipeline` containing `PolynomialFeatures` transformer, followed by a `StandardScaler` and a `LinearSVC`. The example below will be used on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving half circles. Below will be the code:

```
from skelarn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
```

```
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)
```

Additionally, below we can see what the decision boundary of the polynomial SVM classifier looks like on the moons dataset.
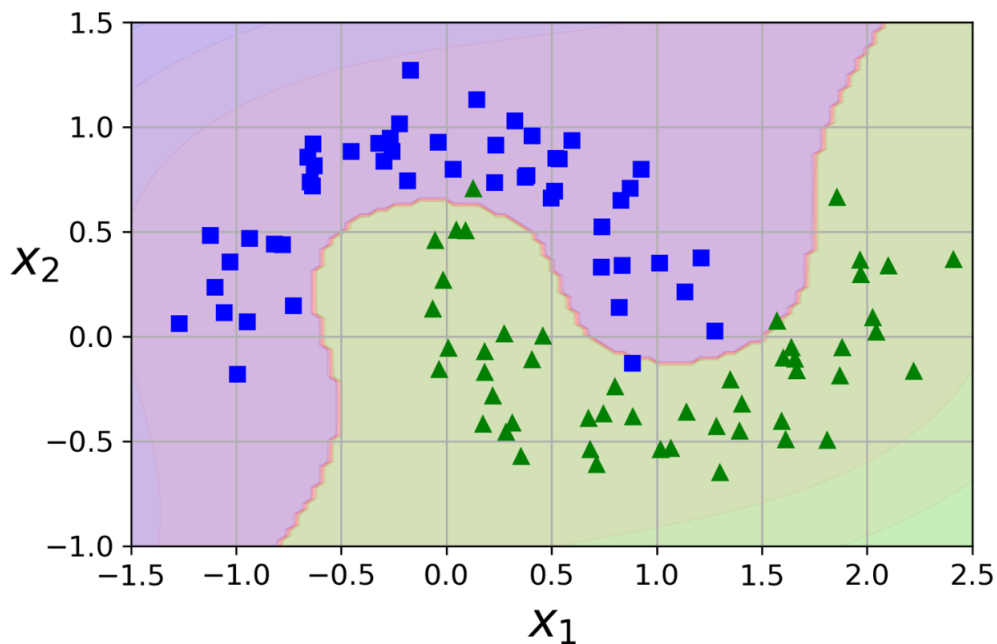
3.3.1   Polynomial Kernel

As seen above adding polynomial features is simple and quite easy to implement. However, at a low polynomial degree, this method does not have the ability to deal with very complex datasets, and with high degree polynomials it creates a huge number of features making model training too slow.

Fortunately, for SVMs, we can use the *kernel trick*. As explained earlier, this trick makes it possible to get the same result as if we had added many polynomial features, without having to actually add them into the model. This trick is already built into the SVC class. Below will be testing 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)
```

In the code above, we are training a polynomial SVM using the kernel trick. Specifically in this case, we are training a 3-degree polynomial SVM classifier.

In the graphic below, we can see the 3-degree polynomial model on the left and on the right there is another 10-degree polynomial kernel. The hyperparameter `coef0` controls how much the model is influenced by high-degree polynomials versus low-degree polynomials.
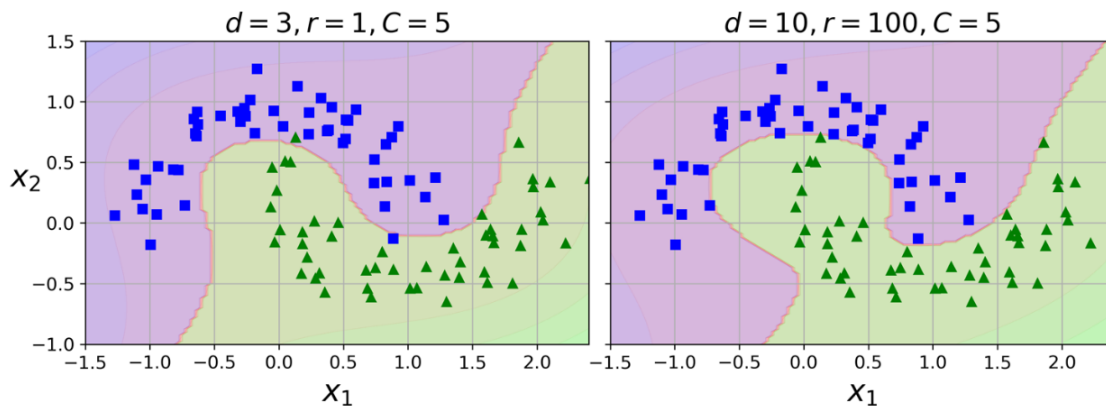
Note:, generally it is common practice to use grid search when trying to find the right hyperparameter values. It is fastest to first do a very coarse grid search, and then a finer grid search around the best values found.

3.3.2 Gaussian RBF Kernel

Once again we can use the *kernel trick* to reduce the computational expensiveness to compute all the additional features if we were to add all the similarity features manually. Below will be the implementation of `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)
```

Below will be a graphic showing different decision boundaries of the Gaussian RBF Kernel for varying values of `gamma` (γ) and `C`. Increasing `gamma` makes the bell-shaped curve narrower. As a result, each instance's range of influence is smaller, in other words, the decision boundary ends

up being more irregular, wiggling around individual instances. Conversely, a small `gamma` value makes the bell-shaped curve wider, therefore, instances have a larger range of influence, and the decision boundary ends up smoother. So γ acts like a regularization hyperparameter: if we have a model that is overfitting, we should reduce γ and if it is underfitting, we should increase γ.
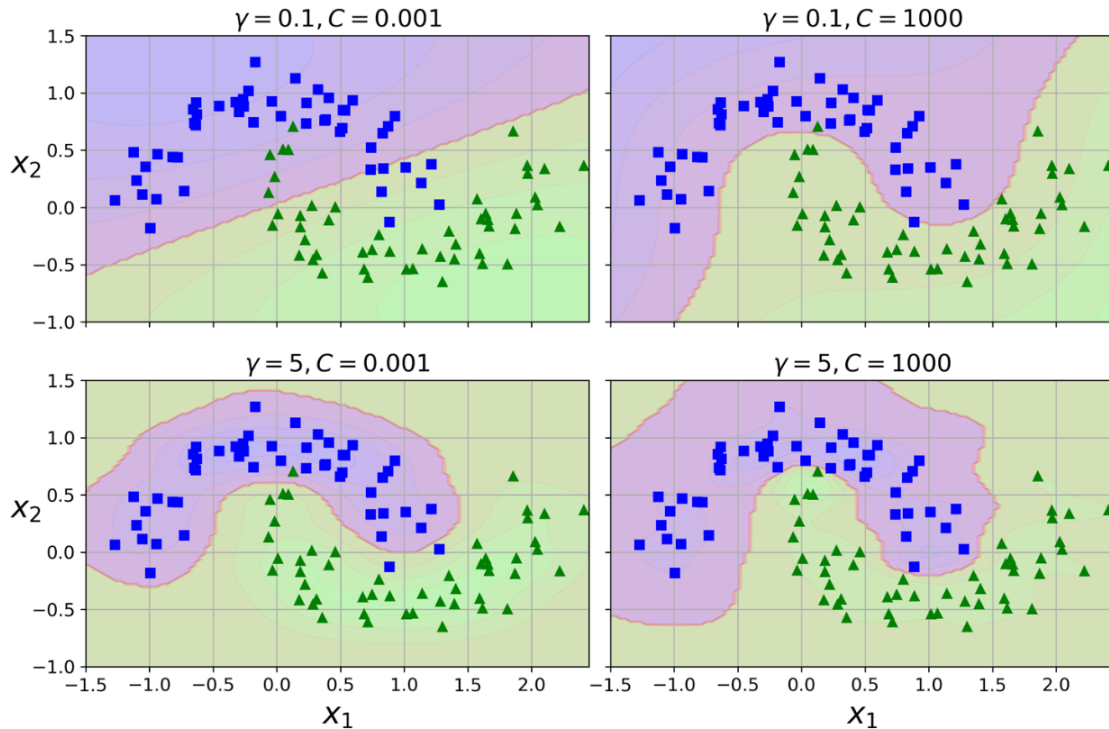
There are other kernels that exist as well, but are used much more rarely. Some kernels are for specific data structures. For example, *String Kernels* are sometimes used when classifiying documents or DNA sequences (e.g., using the *string subsequence kernel* or kernels based on the *Levenshtein distance*).

As a Tip:, one should always try the linear kernel first (remember that `LinearSVC` is much faster than `SVC(kernel="linear")`), especially is the training set is very large or if it has many features. Additionally, we can also try the Gaussian RBF kernel is the training set is not too large. Finally, it is never a bad idea to try and use a kernel specialized whatever data structure that the training set may have.

4   Decision Trees

Decision Trees are a versatile Machine Learning algorithm, that can outperform both classification and regression tasks, and even multioutput tasks. This algorithm is very powerful, and is capable of fitting very complex datasets. Decision Trees are also the fundamental building blocks of a Random Forest algorithm (Covered next section), which is one of the most powerful machine learning algorithms available today.

In this section, we will cover how to train, visualize, and make predictions with Decision Trees

(Using both Regression and classification).

4.1 Training and Visualize a Decision Tree

One of the easiest ways to understand *Deicision Trees* is to build one and take a look how it makes predictions. Below will be some code that trains a `DecisionTreeClassifier` on the iris dataset in Scikit-Learn:

```python
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # Petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X, y)
```

After running this code we can then visualize the Decision Tree by a graph format shown below:
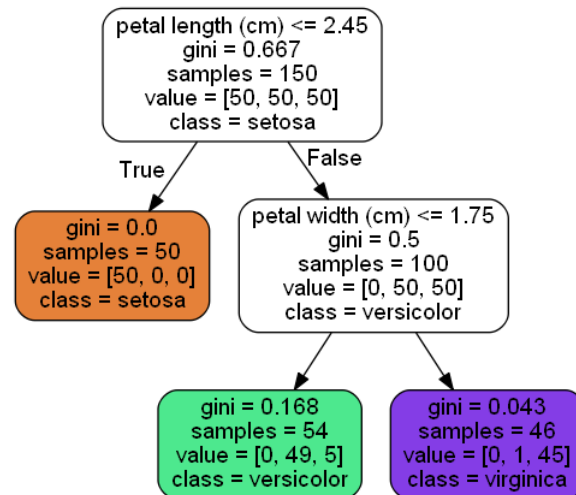
4.2 Making Predictions

From the tree above we can show a logical process to classifying an iris flower.

For example, lets say that we find an iris flower and we want to classify it, the steps will be as follows:

- First we start at the *root node* (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm.

- Second, if it is, then we move down to the root's left child node (depth 1, left)

- Third, we notice that in this case, this is a *leaf node* (i.e., does not have any child nodes), so it is able to predict the class of the iris. Which in this case, case classifies the iris flower as an *iris setosa*

Note: One of the great qualities about Decision Trees is that they require very little data preparation. I.e., they do not require feature scaling or centering at all.

Node's Attributes:

- `samples` attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), and of those 100, 54 have a petal width smaller than 1.75 cm (depth 2, left).

- `value` attribute tells us how many training isntances of each class this node applies to: for example, the bottom-right node applies to 0 *iris setosa*, 1 *iris versicolor*, and 45 *iris virginica*.

- `gini` attribute measures the *impurity*: a node is "pure" (`gini`=0) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies to only *iris setosa* training instances, it is pure and its gini score is 0.

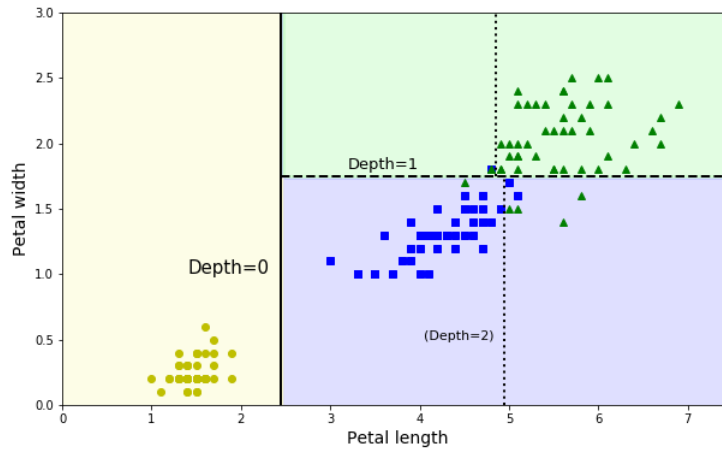Below will be the equation to compute the Gini Score:

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

In the equation above, $p_{i,k}$, is the ratio of class k instances among the training instances in the ith node

Where we can compute the depth-2 left node gini attribute by the following:

$$1 - \left(\frac{0}{54}\right)^2 - \left(\frac{49}{54}\right)^2 - \left(\frac{5}{54}\right)^2 \approx 0.168$$

We can also view the decision boundaries of this Decision Tree through plots made in `matplotlib`.

4.3 Estimating Class Probabilities

A Decision Tree can also estimate the probability that an instance belongs to a particular class k. First it traverses the tree to find the leaf node for this instance, and then it returns the ration of training instances of class k in this node. For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth-2 left node, so the Decision Tree should output the following probabilities: 0% for Iris setosa $\left(\frac{0}{54}\right)$, 90.7% for Iris versicolor $\left(\frac{49}{54}\right)$, and 9.3% for Iris virginica $\left(\frac{5}{54}\right)$. And if you ask it to predict the class, it should output Iris versicolor (class 1) because it has the highest probability.

We can verify this with the following code:

```
tree_clf.predict_proba([[5, 1.5]])
'''
Output:
array([[0.        , 0.90740741, 0.09259259]])
'''


tree_clf.predict([[5, 2.5]])
'''
Output:
array([1])
'''
```

4.4 The CART Training Algorithm

Scikit-Learn uses the *Classification and Regresstion Tree* (CART) algorithm to train Decision Trees. This algorithm works by first splitting the training set into two subsets using a single feature k and a threshold t_k (e.g., "Petal Length *leq* 2.45 cm). It chooses k and t_k by searching

for the pair (k, t_k) that produces the "purest" subsets (weighted by their size). The function for this can be seen below:

$$\mathbf{J}(k, t_k) = \frac{m_{left}}{m}G_{left} + \frac{m_{right}}{m}G_{right}$$

$$\text{where} \quad \begin{cases} G_{\text{left/right}} & \text{Measures the impurity of the left/right subset} \\ m_{\text{left/right}} & \text{Is the number of instances in the left/right subset} \end{cases}$$

Once the CART algorithm has sucessfully split the trianing set in two, it splits the subsets using the same logic then the sub-subsets, and so on. The algorthm stops either when it reaches the maximum depth defined by the modeller, or if it cannot find a split in the subset of data that will reduce the impurity.

Warning: It is important to know that the CART algorithm is a *greedy algorithm*, i.e., it will greedily search for an optimum split at the top level, and then repeats the process at each subsequent level. It **DOES NOT** check whether or not thye split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a solution that's reasonably good but not guaranteed to be optimal. (Note that finding the optimal tree is known to be an *NP-Complete* problem, which is why we must settle with a "reasonably good" solution.)

4.5   Gini Impurity or Entropy?

In Scikit-Learn, the Gini impurity is the default measure used in training Decision Tree Classifiers. However, we do have the option to select *entropy*, which is done by setting the `criterion` hyperparameter in a `DecisionTreeClassifier` to `"entropy"`. Entropy has a similar meaning to the Gini impurity, in that, if entropy is zero for a given set, then all instances in that set belong to the same class. Entropy can be calculated using the equation below:

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k}log_2(p_{i,k})$$

In most cases the deicison between using Entropy vs. Gini impurity does not make a big difference: they tend to usually lead to having similar trees. Gini impurity is slightly faster to compute, so it tends to be a better default in that sense. In the cases when these two methods do tend to differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.

4.6   Regularization Hyperparameters

It is important to know that Decision Trees do make many assumptions about the training data (as opposed to linear models (i.e., assume data is linear)). Therefore, if Decision trees are left unconstrained the tree structure will adapt itself to the training data, and likely severly overfit it.

Such a model with these type of traits are called *nonparametric models*, not because the model itself does not have any parameters, but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. On the other hand, a *parametric model*, such as a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but also increasing the risk of underfitting).

To reduce the risk of overfitting the training data, we need to restrict the Decision Tree's freedom during training. The regularization hyperparameters depend on the algorithm used, but generally we can at least restrict the maximum depth of the Decision Tree. In Scikit-Learn, this is controlled by the `max_depth` hyperparameter (The default is `None`, which means unlimited). Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.

The `DecisionTreeClassifer` class has a few other parameters that similarly restrict the shape of the Decision Tree. These other hyperparameters will be listed below:

- `min_samples_split`: The minimum number of samples a node must have before it can split

- `min_samples_leaf`: The minimum number of samples a leaf node must have

- `min_weight_fraction_leaf`: Same as `min_samples_leaf` but expressed as a fraction of the total number of weighted isntances

- `max_leaf_nodes`: The maximum number of leaf nodes

- `max_features`: The maximum number of features that are evaluated for splitting at each node

NOTE: Other algorithms work by first training the Decision Tree without restrictions, then *pruning* (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistcal tests, such as the χ^2 test (chi-squared test), are used to estimate the probability that the improvement is purely a result of chance (which is called the *null hypothesis*). If the p-value, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. the pruning continues until all unnecessary nodes have been pruned.

Finally, below will be an example of an unrestricted Decision tree and regularized Decision tree on the moons dataset in Scikit-Learn. We can see that the model that does not have any regularization will have a much harder time generalizing as compared to the model with regularization.
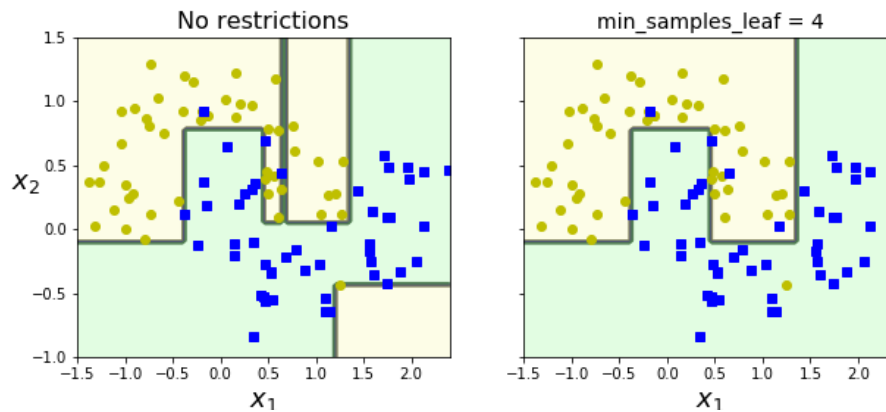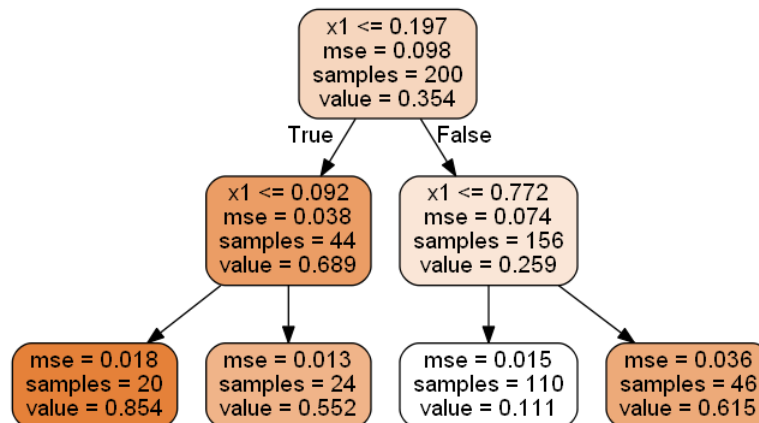
4.7 Regression

Decision Trees are capable of performing regression tasks as well. Below we will use Scikit-Learn's `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with a `max_depth=2`:

```python
from sklearn.tree import DecisionTreeRegressor

# Quadratic training set + noise
np.random.seed(42)
m = 200
X = np.random.rand(m, 1)
y = 4 * (X - 0.5) ** 2
y = y + np.random.randn(m, 1) / 10

tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg.fit(X, y)
```

The resulting tree can be seen below:

We can see that the regression tree looks very similar to the tree that we built earlier. The main difference now is that instead of predicting a class in each node, we are now predicting a value instead. Additionally, we can use the same strategy as before when traversing the classification based tree (For example, if we wanted to find the node for an input value of $x_1 = 0.6$).

This model's predictions are represented on the left in figure below. If you set `max_depth`=3, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.
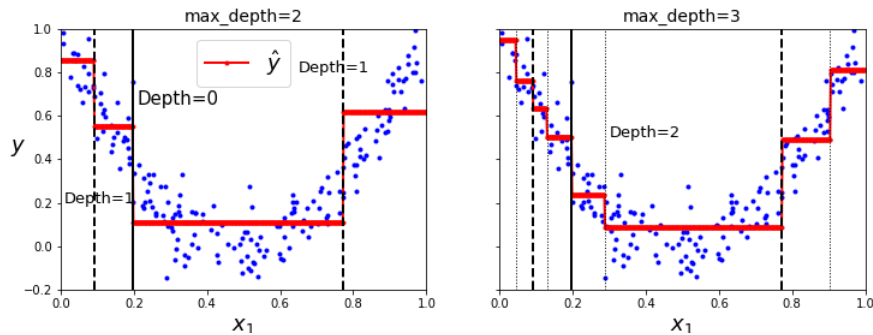

The CART algorithm works similarly, except that instead of trying to split the training set in a way that minimizes impurity, it tries to split the training set in a way that minimizes the *mean squared error* (MSE). Below will be the cost function that the CART algorithm tries to minimize:

$$\mathbf{J}(k, t_k) = \frac{m_{left}}{m} \text{MSE}_{left} + \frac{m_{right}}{m} \text{MSE}_{right}$$

$$\text{where} \quad \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{node} - y^{(i)} \right)^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

Additionally, just like classification tasks, Decision Trees are prone to overfitting when dealing with regression tasks. From the figure below we can see that without any regularization (i.e., using default hyperparameters), we get a model that severely overfits the training data. However, even just setting `min_samples_leaf`=10 results in a model that is much more reasonable and looks like it will generalize much better.


36
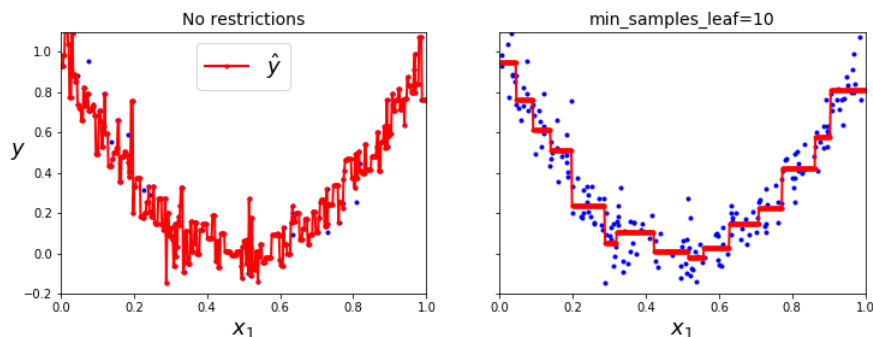
4.8 Instability

Although Decision Trees are a very powerful algorithm, they do have their own limitations. Firstly, Decision Trees tend to use orthogonal decision boundaries (i.e., splits are perpendicular to an axis), which makes them sensitive to training set rotations. For example, in the figure below we can see that the left-hand plot is easily split by a decision tree, however, if we rotate the data 45 degrees, the decision boundary looks to be unnecessarily complex. This then causes further issues with the model not being able to generalize well. One thing to fix this issue is to use Principal Component Analysis (PCA) which often results in better orientation in the data.
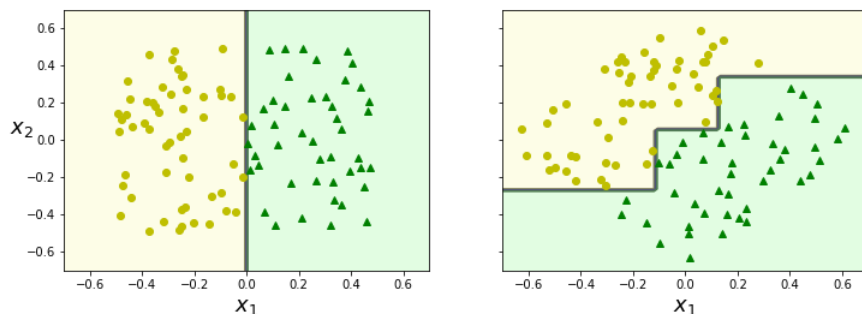
However, the main issue with Decision Trees is that they tend to be very sensitive to small variations in the training data. For example, if we were to remove the widest *Iris Veriscolor* from the iris training set and train a new Decision Tree, we will get a model that is totally different from the model we initially trained. Below will be the plot of the new decision boundaries of the decision tree that does not include the widest iris.
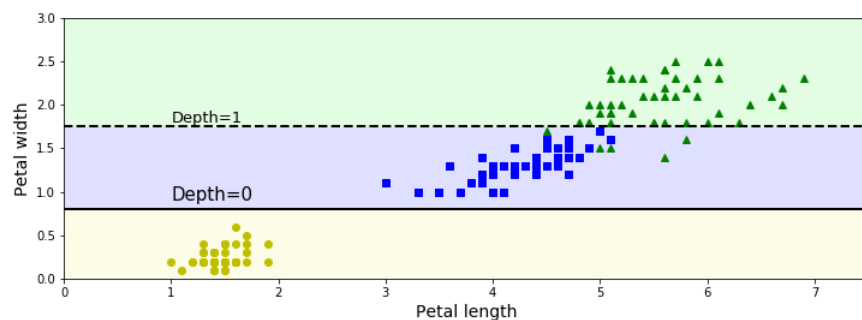
As we will see in the next section, Random Forests can limit this instability affect by averaging predictions over many trees.

5 Random Forest

In this section we will be discussing the *Random Forest* algorithm. As mentioned previously, a Random Forest is an ensemble of individual decision trees trained via the bagging or pasting method (Mostly bagging). In Scikit-Learn, there are both `RandomForestClassifier` and

`RandomForestRegressor` depending on the task we need to complete at hand.

The code below will train a Random Forest classifier with 500 trees using all available CPU cores (Each tree limited to 16 nodes):

```python
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

X, y = make_moons(n_samples=500, noise=0.30)
X_train, X_test, y_train, y_test = train_test_split(X, y)

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                 n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

For the most part a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how the trees are grown), plus the ability to control the ensemble of the trees as well.

The Random Forest Algorithm introduces more randomness when growing trees; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features. Therefore the algorithm results in greater tree diversity, which trades a higher bias for a lower variance, generally returning a overall better model.

5.1   Feature Importance

Conveniently, Random Forests are also able to make it easy to measure the relative importance of each feature given a dataset. Scikit-Learn measures a feature's important by looking how much the tree nodes that use the feature reduce impurity on average (across all trees in the forest). More specifically, it is a weighted averafe, where each node's weight is equal to the number of training samples that are associated with it.

Scikit-Learn computes the score automatically and then scales the result fo that the sum of all feature importances are equal to 1. To access these results is fairly straight forward by calling the `feature_importances_` variable. For example, below will be a code block trainig a `RandomForestClassifier` on the iris dataset and outputs each feature's importance.

```python
from sklearn.datasets import load_iris

iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, njobs=-1)
rnd_clf.fit(iris["data"], iris["target"])
```

```python
for name,score in zip(iris["feature_names"], rnd_clf.feature_importances_):
    print(name, score)

'''
Output:

sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
'''
```

Now we can see that it seems that the most important features are petal length and petal width with 44% and 42% important, respectively.

6 Dimensionality Reduction

Many problems in Machine Learning involve thousands or even millions of features for each training instance. Having this many features often make training very slow and also make it much harder to find a good solution as well (Due to the *Curse of Dimensionality*, will be explained more later).

However, we are often able to reduce the number of features quite considerably, turning a rather complex problem into something more simple and/or manageable. Although, we should take care in how we reduce dimensions as we do lose information with each dimension that we remove. So, if training isn't too slow, we should always first try to train the system with the original dataset before considering reducing dimensionality.

In addition, there are some cases where reducing dimensionality of the training data may filter out some noise and thus result in higher performance, but in general it will not increase performance and will only decrease the amount of time it will take to train the model.

6.1 Curse of Dimensionality

Given that we are living in 3-dimensional space (4th including time), our understanding of anything above 3-dimensions is incredibly hard to grasp. It also turns out that many things behave very differently in high-dimensional space (Please see below).

Expected Distance Between 2 Randomly Chosen Points in Different Dimensions

- **Unit Square**

  - Distance on Average is roughly 0.52

- **Unit 3D Cube**

– Distance on Average is roughly 0.66

- **1 Million Dimensional Hypercube**

  – Distance on Average is roughly 408.25 ($\sqrt{\frac{1,000,000}{6}}$)

Therefore, as a result of above, we can see that high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. This also means that a new instance will likely be far away from any training instance, making predictions much less reliable compared to a lower dimensional model.

In short, we see that the more dimensions we add, the higher the risk of overfitting the data with our model.

6.2 Main Approaches for Dimensionality Reduction

Before taking a look at specific algorithms to reduce dimensionality, it is important to first take a look at the two main approaches in reducing dimensionality: *Projection* and *Manifold Learning*.

6.2.1 Projection

In many cases we will find that the training data we will be using is not spread out uniformly across all dimensions. Many features will tend to be constant, while others may be highly correlated. Therefore, we find that many of the training instances lie within (or close to) a much lower-dimensional subspace of the high-dimensional space.

Generally, projection works best when the data we are trying to project to lower dimensions does not have too many twists and turns that may cause points to overlap each other when brought down to lower dimensions. A good example to see this will be using the *Swiss Roll* dataset that can be found in Scikit-Learn's library.

6.2.2 Manifold Learning

In the previous section the *Swiss Roll* data set was mentioned, this is an example of a 2D manifold. In layman's terms, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d-dimensional manifold is a part of an n-dimensional space where $(d < n)$ that locally resembles a d-dimensional hyperplane. In the case of the swiss roll, $d = 2$ and $n = 3$: thus, it locally resembles a 2D plane, but it is rolled in the third dimension.

In practice, many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie; this is called *manifold learning*. The prior relies on the *manifold assumption*, also known as the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. Empirically, this assumption is very commonly observed.

6.3 Principal Compenents Analysis (PCA)

Principal Components Analysis or PCA, for short, is by far one of the most-popular dimensionality reduction algorithms. In its most basic terms, PCA identifies a hyperplane that lies closest to the data, and then it projects the data onto it.

Principal Components

When projecting training data onto a lower-dimensional plane, it is important that we are choosing the correct hyperplane. The algorithm PCA identitifies the axis that accounts for the largest amount of variance in the training set (Here variance can be thought of the amount of information captured from a single hyperplane that the training data is laying on).

PCA finds as many axes as there are dimensions in the dataset, however, it does this is such a way where it finds the planes that contain the most amount of variance first and then they decrease in each component (Note, that each component is orthogonal to each of the previous axes).

Here we can say that the ith axis is the ith *principal component* (PC) of the data. It is important to know that for each principal component, PCA finds a zero-centered unit vector pointing in the direction of the PC. Since two opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA are inherently not stable. For example, if we were to peturb the training data slightly and run PCA on the data, the unit vectors may point in the opposite direction as the original vectors in the non perturbed dataset. However, in the grand scheme of things, this unstable chracteristic generally does not matter as the unit vectors, regardless of direction, tend to lie on the same axes, therefore, the plane will be the same.

PCA uses a standard matrix decomposition technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices $\mathbf{U\Sigma V^\intercal}$, where $\mathbf{V}$ contains the unit vectors that define all the principal components that we are looking for, as shown below:

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c_1} & \mathbf{c_2} & \cdots & \mathbf{c_n} \\ | & | & & | \end{pmatrix}$$

Below will be using NumPy's `svd()` function to obtain all the principal components of some given matrix $\mathbf{X}$:

```python
import numpy as np

X_centered = X - X.mean(axis=0) # Centering Matrix
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0] # First Principal Component
c2 = Vt.T[:, 1] # Second Principal Component
```

Important: PCA assumes that the dataset is centered around the origin. However, in Scikit-Learn's PCA classes, centering is automatically taken care of for you. But if you implement PCA yourself, or if you are using other libraries, **do not** forget to center the data first.

Projecting Down to d Dimensions

Once all principal components of a datset are identified, we can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting the first few hyperplanes will ensure that the projection we are doing will preserve the most variance as possible.

To project the training set onto the hyperplane and obtain a reduced $\mathbf{X}_{d-proj}$ of dimensionality d, compute the matrix multiplication of the training set $\mathbf{X}$ by the matrix $\mathbf{W}_d$, defined as the matrix the matrix containing the first d columns of $\mathbf{V}$, as shown below:

$$\mathbf{X}_{d-proj} = \mathbf{X}\mathbf{W}_d$$

The following Python code will be projecting the training set ($\mathbf{X}$) onto the plane defined by the first two principal components:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

Thus, that is how we can manually reduce dimensionality of a dataset down to any number of dimensions that we desire to acquire.

Using Scikit-Learn for PCA

Scikit-Learn's `PCA` class uses SVD decomposition to implement PCA (as we have manually done before). The following code in this section will be covering how to use Scikit-Learns built-in PCA function.

First we can import PCA in the following way (note that in Scikit-Learn centering is done automatically):

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2) # Only want the first 2 principal components
X2D = pca.fit_transform(X) # Projecting X down to 2 dimensions
```

Further, after fitting the `PCA` transformer to the dataset, the attribute `components_` holds the transpose of $\mathbf{W}_d$ (e.g., the unit vector that defines the first principal component is equal to `pca.components_.T[:, 0]`)

A useful variable that is provided by Scikit-Learn's `PCA` class is the `explained_variance_ratio_` variable. This variable indicates the proportion of the dataset's variance that lies along each principal component. We can all the variable in the following way:

```python
pca.explained_variance_ratio_

'''
Example output for 2 principal compoenents would be:
array([0.84248607, 0.14631839])
'''
```

Here that example output is telling us that 84% of the dataset's variance lies along the first PC, and 14% lies along the second PC. This leaves approximately 2% of the variance to be in the rest of the PCs.

Choosing the Right Number of Dimensions

In practice, we should not be arbitrarily choosing the numbers that we will want to reduce our dataset down to. Luckily, in Scikit-Learn's PCA class we are able to define the argument `n_components` to a number between 0 and 1. For example, if we set the argument, `n_components` equal to 0.95, then we will be asking to return the number of components that preserve 95% of the data's variance. Below will be an example of implementing this using Scikit-Learn:

```python
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
# Here X_train is just some random variable that is used for example
# purposes.
```

Note: A very good dataset to see the benefits of PCA is applying it to the MNIST dataset (Can be found in Scikit-Learn) that has about 60,000 images of handwritten notes. One will find that using PCA we can preserve 95% of the variance in that dataset even though we reduce the total dimensions down to just over 150 from 784 features (Almost less than 20% of the original size of the dataset).

6.4   Kernel PCA

6.5   Locally Linear Embedding (LLE)

6.6   Other Dimensionality Reduction Techniques

7   Unsupervised Learning

7.1   Clustering

7.2   Gaussian Mixtures