en ligne, lorsque vous devez effectuer une PCA à la volée, chaque fois que de nouvelles observations arrivent. La PCA randomisée est utile lorsque vous voulez réduire considérablement la dimension et que le jeu de données tient en mémoire : dans ce cas, elle est beaucoup plus rapide qu'une PCA ordinaire. Enfin, la PCA à noyau est utile pour les jeux de données non linéaires.

7. Intuitivement, un algorithme de réduction de dimension donne de bons résultats s'il élimine beaucoup de variables du jeu de données sans perdre trop d'information. Une façon d'évaluer cette perte d'information consiste à appliquer la transformation inverse et à mesurer l'erreur de reconstruction. Cependant, tous les algorithmes de réduction de dimension ne proposent pas de transformation inverse. Si vous effectuez une réduction de dimension en tant qu'étape préparatoire à un autre algorithme d'apprentissage automatique (p. ex. un classificateur de forêt aléatoire), vous pouvez aussi mesurer simplement les résultats de ce second algorithme. Si la réduction de dimension n'a pas perdu trop d'information, alors l'algorithme devrait donner d'aussi bons résultats qu'en travaillant directement sur le jeu de données d'origine.
8. Il peut être tout à fait intéressant d'enchaîner deux algorithmes de réduction de dimension différents. Pour en donner un exemple courant, on peut utiliser une PCA pour se débarrasser rapidement d'un grand nombre de dimensions inutiles, puis appliquer un autre algorithme de réduction de dimension beaucoup plus lent, comme LLE. Cette approche en deux étapes donnera vraisemblablement les mêmes résultats qu'en utilisant uniquement LLE, mais en ne consommant qu'une fraction du temps de calcul.

Pour les solutions des exercices 9 et 10, reportez-vous aux notebooks Jupyter publiés sur *https://github.com/ageron/handson-ml*.

Annexe B

Liste de contrôle de projet de Machine Learning

Cette liste de contrôle vous guidera tout au long de vos projets de Machine Learning. Ceux-ci se dérouleront en huit étapes principales :

1. Cerner le problème et rechercher une vision d'ensemble.
2. Récupérer les données.
3. Explorer les données pour mieux les comprendre.
4. Préparer les données pour mieux exposer leurs structures sous-jacentes aux algorithmes d'apprentissage automatique.
5. Essayer beaucoup de modèles différents et constituer une liste restreinte des meilleurs d'entre eux.
6. Régler finement les modèles et les combiner en une solution performante.
7. Présenter votre solution.
8. Lancer, surveiller et maintenir votre système.

Bien sûr, vous pouvez librement adapter cette liste à vos besoins.

CERNER LE PROBLÈME ET RECHERCHER UNE VISION D'ENSEMBLE

1. Définir l'objectif professionnel
2. Comment votre solution sera-t-elle utilisée ?
3. Y a-t-il actuellement des solutions ou contournements ?
4. Comment formuleriez-vous le problème (apprentissage supervisé / non supervisé, en ligne / groupé) ?
5. Comment la qualité des résultats sera-t-elle mesurée ?

6. La mesure de performance choisie s'accorde-t-elle à votre objectif professionnel ?
7. Quelle est la performance minimale permettant d'atteindre votre objectif professionnel ?
8. Existe-t-il des problèmes comparables ? Pouvez-vous profiter de cette expérience ou de ces outils ?
9. Existe-t-il une expertise humaine sur la question ?
10. Comment résoudriez-vous le problème manuellement ?
11. Faire la liste des hypothèses faites par vous-même ou par d'autres personnes jusqu'ici.
12. Vérifier si possible ces hypothèses.

OBTENIR LES DONNÉES

Remarque : automatiser autant que possible pour pouvoir récupérer facilement les nouvelles données.

1. Faire la liste des données nécessaires et de la quantité nécessaire.
2. Trouver comment récupérer les données et le noter par écrit.
3. Calculer la place nécessaire.
4. Vérifier les obligations légales et obtenir au besoin les autorisations.
5. Obtenir les droits d'accès.
6. Créer un environnement de travail (avec un espace de stockage suffisant).
7. Récupérer les données.
8. Convertir les données dans un format aisément manipulable (sans modifier les données elles-mêmes).
9. S'assurer que les informations sensibles sont supprimées ou protégées (anonymisées).
10. Vérifier la taille et le type des données (séries temporelles, échantillons, données géographiques, etc.).
11. Constituer un jeu de test, le mettre de côté et ne jamais le consulter (pas d'espionnage des données !).

EXPLORER LES DONNÉES

Remarque : pour ces étapes, essayez d'obtenir l'avis d'un spécialiste du domaine.

1. Créer une copie des données pour cette exploration (ou prélever un échantillon de taille raisonnable si nécessaire).
2. Créer un notebook Jupyter pour conserver une trace de votre exploration des données.

3. Étudier les caractéristiques de chaque variable :
   - Nom
   - Type (qualitative, numérique à valeurs entières ou réelles, bornée ou non, textuelle, structurée, etc.)
   - Taux de données manquantes
   - Présence d'aléas et nature de ceux-ci (bruit aléatoire, données aberrantes, erreurs d'arrondi, etc.)
   - Utilité pour la tâche ?
   - Type de distribution (gaussienne, uniforme, logarithmique, etc.)
4. Pour les tâches d'apprentissage supervisé, identifier la (ou les) variable(s) cible.
5. Visualiser les données.
6. Étudier les corrélations entre variables.
7. Étudier la façon de résoudre le problème manuellement.
8. Identifier les transformations qu'il pourrait être intéressant d'appliquer.
9. Identifier des données supplémentaires qui pourraient être utiles (revenir à l'étape « Obtenir les données »).
10. Noter les enseignements retenus.

PRÉPARER LES DONNÉES

Remarques :

- Travailler sur des copies des données (conserver le jeu d'origine intact)
- Écrire des fonctions pour toutes les transformations de données réalisées :
  - pour préparer aisément les données lorsque vous les actualisez,
  - pour pouvoir utiliser ces transformations dans vos projets futurs,
  - pour nettoyer et préparer le jeu de test,
  - pour nettoyer et préparer les nouvelles données une fois que votre solution est en production,
  - pour pouvoir transformer aisément vos choix de préparation en hyperparamètres.

1. Nettoyer les données :
   - corriger ou supprimer les données aberrantes (facultatif) ;
   - remplacer les données manquantes (par des zéros, la moyenne, la médiane, etc.) ou bien supprimer les observations ou les variables correspondantes.
2. Sélectionner les variables (facultatif) :
   - en éliminant celles qui ne fournissent pas d'information utile pour cette tâche.

3. Transformer au besoin les variables :
   – transformer certaines variables continues en variables discrètes (ou qualitatives) ;
   – décomposer les variables (p. ex. variables qualitatives, date et heure, etc.) ;
   – appliquer certaines transformations prometteuses aux variables (p. ex. log(x), sqrt(x), x^2, etc.) ;
   – combiner plusieurs variables en une nouvelle variable plus intéressante.
4. Recalibrer les variables par une normalisation ou une transformation min-max.

DRESSER UNE LISTE RESTREINTE DE MODÈLES PROMETTEURS

Remarques :

- Si les données sont volumineuses, vous pouvez envisager de prélever aléatoirement des jeux d'entraînement plus petits pour pouvoir entraîner en un temps raisonnable de nombreux modèles différents (sachez que cela pénalise les modèles complexes tels que les grands réseaux neuronaux ou les forêts aléatoires).
- Là encore, veiller dans toute la mesure du possible à automatiser ces tâches.

1. Entraîner avec les paramètres usuels de nombreux modèles rapides et basiques de différentes catégories (ex. linéaire, bayésien naïf, SVM, forêt aléatoire, réseau neuronal, etc.).
2. Mesurer et comparer les résultats obtenus.
   – Pour chaque modèle, utiliser une validation croisée à *N* passes et calculer la moyenne et l'écart-type de la mesure de performance sur chacune des *N* passes.
3. Rechercher les variables les plus significatives pour chaque algorithme.
4. Analyser la nature des erreurs faites par les modèles.
   – Quelles données un humain aurait-il utilisées pour éviter ces erreurs ?
5. Effectuer une rapide étape de sélection et d'ingénierie de variables.
6. Effectuer rapidement une ou deux itérations supplémentaires sur les cinq étapes précédentes.
7. Dresser une liste restreinte des trois à cinq modèles les plus prometteurs, en choisissant de préférence des modèles ayant réalisé des erreurs de nature différente.

RÉGLER FINEMENT LE SYSTÈME

Remarques :

- Pour cette étape, il sera souhaitable d'utiliser autant de données que possible, surtout pour la phase finale de vos réglages.
- Comme toujours, automatisez autant que possible.

1. Régler les hyperparamètres par validation croisée.
   – Traiter les choix de transformation de données comme des hyperparamètres, surtout en cas de doute sur leur utilité (p. ex., vaut-il mieux remplacer les valeurs manquantes par des zéros, par la valeur médiane, ou supprimer simplement les lignes correspondantes ?
   – Préférer une recherche aléatoire à une recherche par quadrillage, sauf s'il y a très peu de valeurs d'hyperparamètres à explorer. Si l'entraînement est très long, une optimisation bayésienne sera peut-être préférable (p. ex. en utilisant un processus gaussien *a priori*[109]).
2. Essayer les méthodes d'ensemble : combiner vos meilleurs modèles donnera souvent de meilleurs résultats que de les exécuter individuellement.
3. Après avoir obtenu un modèle final digne de confiance, mesurer ses performances sur le jeu de test pour estimer l'erreur de généralisation.

N'ajustez plus votre modèle après avoir mesuré l'erreur de généralisation, car vous ne feriez que surajuster le jeu de test.

PRÉSENTER VOTRE SOLUTION

1. Faire une synthèse écrite de ce qui a été fait.
2. Créer une présentation agréable.
   – Prendre soin de commencer par en souligner les grandes lignes.
3. Expliquer pourquoi votre solution répond à l'objectif professionnel fixé.
4. Penser à présenter les points intéressants découverts en cours de route.
   – Décrire ce qui a marché et ce qui n'a pas fonctionné.
   – Faire la liste des hypothèses faites et des limitations de votre système.
5. Veiller à communiquer vos résultats clés de manière visuellement attractive ou sous forme d'affirmations faciles à retenir (p. ex. : « le revenu médian est le principal prédicteur des prix immobiliers ».

109. « Practical Bayesian Optimization of Machine Learning Algorithms » (optimisation bayésienne pratique des algorithmes d'apprentissage automatique), J. Snoek, H. Larochelle, R. Adams (2012), *https://goo.gl/PEFfGr*

```
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

Dans cet exemple, le transformateur a un hyperparamètre `add_bedrooms_per_room`, qui a la valeur `True` par défaut (il est souvent utile de fournir des valeurs par défaut raisonnables). Cet hyperparamètre vous permettra de vérifier aisément si l'ajout de cette variable améliore ou non l'algorithme d'apprentissage automatique. Plus généralement, vous pouvez ajouter un hyperparamètre pour surveiller toute étape de préparation des données dont vous n'êtes pas sûr à 100 %. Plus vous automatiserez ces étapes de préparation des données, plus vous pourrez tester de combinaisons, augmentant ainsi vos chances d'en trouver une qui fonctionne très bien (tout en économisant beaucoup de temps).

2.5.4 Recalibrage des variables

L'une des transformations les plus importantes que vous devez appliquer à vos données est le *recalibrage des variables* (en anglais, *feature scaling*). À peu d'exceptions près, les algorithmes d'apprentissage automatique ne fonctionnent pas très bien lorsque les variables numériques en entrée ont des échelles très différentes. C'est le cas pour ces données immobilières : le nombre total de pièces s'étage de 6 à 39 320, tandis que le revenu moyen ne va que de 0 à 15. Notez qu'il n'est pas nécessaire en général de recalibrer les valeurs cibles.

Il existe deux moyens courants de mettre toutes les variables à la même échelle : la transformation min-max et la normalisation.

La *transformation min-max* (en anglais, *min-max scaling*, ou parfois *normalization*) est assez simple : les valeurs sont décalées et recalibrées afin qu'elles se situent toutes entre 0 et 1. Il suffit pour cela de leur soustraire la valeur minimum, puis de diviser par *max* – *min*. Scikit-Learn fournit pour cela un transformateur appelé `MinMaxScaler`. Il possède un hyperparamètre `feature_range` (intervalle de variation) qui vous permet de changer la fourchette, si 0-1 ne vous convient pas pour une raison ou une autre.

La *normalisation* (en anglais, *standardization*) est assez différente : elle consiste tout d'abord à soustraire la valeur moyenne (de sorte que les valeurs normalisées auront toujours une moyenne nulle), puis à diviser par l'écart-type afin que la distribution qui en résulte ait une variance égale à 1. Contrairement à la transformation min-max, la normalisation ne limite pas les valeurs à un intervalle donné, ce qui pourrait être un problème pour certains algorithmes (les réseaux neuronaux, par exemple, attendent souvent une valeur d'entrée comprise entre 0 et 1). Cependant, la normalisation est beaucoup moins affectée par les valeurs aberrantes : supposons par exemple qu'un des districts ait un revenu médian égal à 100 (par erreur), alors la transformation min-max écraserait toutes les autres valeurs situées dans l'intervalle 0-15 pour les réduire à 0-0,15, tandis que la normalisation n'en serait guère affectée. Scikit-Learn fournit un transformateur appelé `StandardScaler` pour la normalisation.

Comme pour toutes les transformations, il est important de définir les changements d'échelle uniquement sur les données d'entraînement, et non sur les données complètes (incluant le jeu de test). Ce n'est qu'à partir de là que vous pouvez les utiliser pour transformer le jeu d'apprentissage et le jeu de test (et les nouvelles données).

2.5.5 Pipelines de transformation

Comme vous pouvez le voir, il y a de nombreuses étapes de transformation des données, et elles doivent être exécutées dans le bon ordre. Heureusement, Scikit-Learn fournit la classe `Pipeline` qui aide à séquencer ces transformations. Voici un petit pipeline pour les variables numériques :

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Le constructeur `Pipeline` reçoit en entrée une liste de paires nom/estimateur définissant une suite d'étapes. Tous les estimateurs à l'exception du dernier doivent être des transformateurs (c'est-à-dire doivent posséder une méthode `fit_transform()`). Les noms sont à votre convenance.

Lorsque vous appelez la méthode `fit()` du pipeline, celle-ci appelle tour à tour la méthode `fit_transform()` de chacun des transformateurs, récupérant la sortie de chacun des appels pour la transmettre à l'appel suivant, jusqu'à ce qu'elle atteigne l'estimateur final dont elle appelle simplement la méthode `fit()`.

Le pipeline expose les mêmes méthodes que l'estimateur final. Dans cet exemple, le dernier estimateur, `StandardScaler`, est un transformateur : par conséquent, le pipeline possède une méthode `transform()` qui applique consécutivement toutes les transformations aux données (il possède également une méthode `fit_transform` qui aurait pu être appelée au lieu d'appeler successivement `fit()` puis `transform()`).

Vous disposez maintenant d'un pipeline pour les valeurs numériques, mais il vous faut aussi appliquer `LabelBinarizer` aux variables qualitatives : comment associer ces transformations en un seul pipeline ? Pour ce faire, Scikit-Learn fournit la classe `FeatureUnion` : vous lui donnez une liste de transformateurs (qui peuvent être des pipelines de transformateurs), et lorsque sa méthode `transform()` est appelée, elle exécute en parallèle la méthode `transform()` de chacun des transformateurs, se met en attente des sorties puis les concatène et renvoie le résultat (et bien sûr l'appel à sa méthode `fit()` déclenche l'appel de la méthode `fit()` de chacun des transformateurs). Voici à quoi peut ressembler un pipeline complet gérant à la fois des variables quantitatives et qualitatives :

```
from sklearn.pipeline import FeatureUnion

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
```

4.7 EXERCICES

1. Quel algorithme d'entraînement de régression linéaire pouvez-vous utiliser si vous avez un jeu d'entraînement comportant des millions de variables ?
2. Supposons que les variables de votre jeu d'entraînement aient des échelles très différentes. Quels algorithmes peuvent en être affectés, et comment ? Comment pouvez-vous y remédier ?
3. Une descente de gradient peut-elle se bloquer sur un minimum local lorsque vous entraînez un modèle de régression logistique ?
4. Tous les algorithmes de descente de gradient aboutissent-ils au même modèle si vous les laissez s'exécuter suffisamment longtemps ?
5. Supposons que vous utilisiez une descente de gradient ordinaire, en représentant graphiquement l'erreur de validation à chaque cycle (ou époque) : si vous remarquez que l'erreur de validation augmente régulièrement, que se passe-t-il probablement ? Comment y remédier ?
6. Est-ce une bonne idée d'arrêter immédiatement une descente de gradient par mini-lots lorsque l'erreur de validation augmente ?
7. Parmi les algorithmes de descente de gradient que nous avons étudiés, quel est celui qui arrive le plus vite à proximité de la solution optimale ? Lequel va effectivement converger ? Comment pouvez-vous faire aussi converger les autres ?
8. Supposons que vous utilisiez une régression polynomiale. Après avoir imprimé les courbes d'apprentissage, vous remarquez qu'il y a un écart important entre l'erreur d'entraînement et l'erreur de validation. Que se passe-t-il ? Quelles sont les trois manières de résoudre le problème ?
9. Supposons que vous utilisiez une régression ridge. Vous remarquez que l'erreur d'entraînement et l'erreur de validation sont à peu près identiques et assez élevées : à votre avis, est-ce le biais ou la variance du modèle qui est trop élevé(e) ? Devez-vous accroître l'hyperparamètre de régularisation α ou le réduire ?
10. Qu'est-ce qui pourrait vous inciter à choisir une régression…
    - ridge plutôt qu'ordinaire ?
    - lasso plutôt que ridge ?
    - elastic net plutôt que lasso ?
11. Supposons que vous vouliez classer des photos en extérieur/intérieur et jour/nuit. Devez-vous utiliser deux classificateurs de régression logistique ou un classificateur de régression softmax ?
12. Implémentez une descente de gradient ordinaire avec arrêt précoce pour une régression softmax (sans utiliser Scikit-Learn).

Les solutions de ces exercices sont données à l'annexe A.

5
Machines à vecteurs de support

Une *machine à vecteurs de support* ou *séparateur à vaste marge* (SVM[62]) est un modèle d'apprentissage automatique à la fois puissant et polyvalent, capable d'effectuer des classifications linéaires ou non linéaires, des régressions et même des détections de données aberrantes. C'est un des modèles les plus prisés en matière d'apprentissage automatique, et tous ceux qui s'intéressent au Machine Learning devraient en avoir dans leur boîte à outils. Ils sont particulièrement bien adaptés à la classification de jeux de données complexes, mais de taille réduite ou moyenne.

Ce chapitre va vous expliquer les concepts clés des SVM, comment les utiliser et comment ils fonctionnent.

5.1 CLASSIFICATION SVM LINÉAIRE

Pour expliquer l'idée fondamentale qui sous-tend les SVM, rien de tel que des images. La figure 5.1 présente une partie du jeu de données Iris qui a été étudié à la fin du chapitre 4. Il est clair que les deux classes peuvent être aisément séparées par une ligne droite (elles sont *linéairement séparables*). Le diagramme de gauche présente les frontières de décision de trois classificateurs linéaires possibles : le modèle dont la frontière de décision est représentée par une ligne à tirets est si mauvais qu'il ne sépare même pas correctement les classes. Les deux autres modèles fonctionnent parfaitement sur le jeu d'entraînement, mais leurs frontières de décision sont si proches des observations que ces modèles ne donneront probablement pas d'aussi bons résultats sur de nouvelles observations. Au contraire, la ligne continue

62. En anglais, cette méthode porte le nom de Support Vector Machine : la deuxième appellation en français permet donc d'utiliser le même acronyme dans les deux langues.

du graphique de droite représente la frontière de décision d'un classificateur SVM : cette ligne ne se contente pas de séparer les deux classes, mais elle reste aussi distante que possible des observations d'entraînement les plus proches. En quelque sorte, un classificateur SVM ajuste le chemin le plus large possible (matérialisé par les deux lignes pointillées parallèles) entre les classes. C'est ce qu'on appelle une *classification à large marge*.

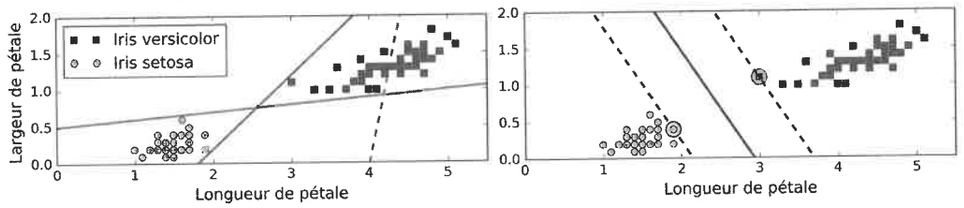


Figure 5.1 – Classification à large marge

Notez que le fait d'ajouter davantage d'observations d'entraînement « hors du chemin » ne modifie aucunement la frontière de décision : elle est entièrement déterminée par les observations situées sur les bords du chemin. Ces observations sont appelées vecteurs de support (elles sont cerclées sur la figure 5.1).

Les SVM sont sensibles aux différences d'échelle des variables, comme vous pouvez le voir sur la figure 5.2 : sur le graphique de gauche, l'échelle verticale est beaucoup plus grande que l'échelle horizontale, c'est pourquoi le chemin le plus large est proche de l'horizontale. Après normalisation (en utilisant par exemple la fonction `StandardScaler` de Scikit-Learn), la frontière de décision paraît bien meilleure (sur le graphique de droite).

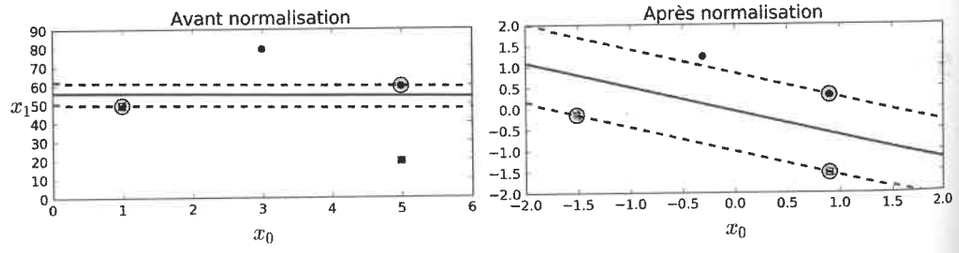


Figure 5.2 – Sensibilité aux échelles des variables

5.1.1 Classification à marge souple

Si nous imposons strictement que toutes les observations soient en dehors du chemin et du bon côté, nous effectuons une *classification à marge rigide* (en anglais, *hard margin classification*). Une classification de ce type présente deux problèmes principaux : tout

d'abord, elle ne fonctionne que si les données sont linéairement séparables ; d'autre part, elle est relativement sensible aux données aberrantes. La figure 5.3 présente le jeu de données Iris auquel on a ajouté simplement une donnée aberrante : à gauche, il est impossible de trouver une marge rigide, tandis qu'à droite la frontière de décision devient très différente de celle obtenue en 5.1 sans donnée aberrante, et ne se généralisera probablement pas bien.

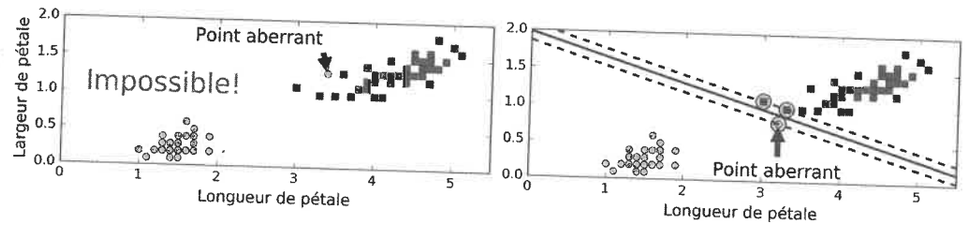


Figure 5.3 – La marge rigide est très sensible aux données aberrantes

Pour éviter ces problèmes, il est préférable d'utiliser un modèle plus accommodant : l'objectif est de trouver un bon équilibre entre conserver un chemin aussi large que possible et limiter les *empiètements de marge* (c.-à-d. le nombre d'observations se retrouvant à l'intérieur du chemin, voire même du mauvais côté). C'est ce qu'on appelle une *classification à marge souple* (en anglais, *soft margin classification*).

Au niveau des classes SVM de Scikit-Learn, vous pouvez contrôler cet équilibre à l'aide de l'hyperparamètre `C` : plus la valeur de `C` est petite, plus le chemin sera large, mais plus vous aurez d'empiètements de marge. La figure 5.4 présente les frontières de décision et les marges de deux classificateurs SVM à marge souple sur un jeu de données non linéairement séparable : à gauche, en utilisant une valeur `C` élevée, le classificateur provoque moins d'empiétements de marge mais aboutit à une marge plus étroite ; à droite, en utilisant une valeur `C` plus faible, la marge est plus large mais beaucoup d'observations se retrouvent à l'intérieur du chemin. Cependant, il est vraisemblable que le second classificateur fournira une meilleure généralisation : de fait, même sur le jeu d'entraînement, il fait moins d'erreurs de prédiction, car la plupart des empiétements de marge sont en fait du bon côté de la frontière de décision.

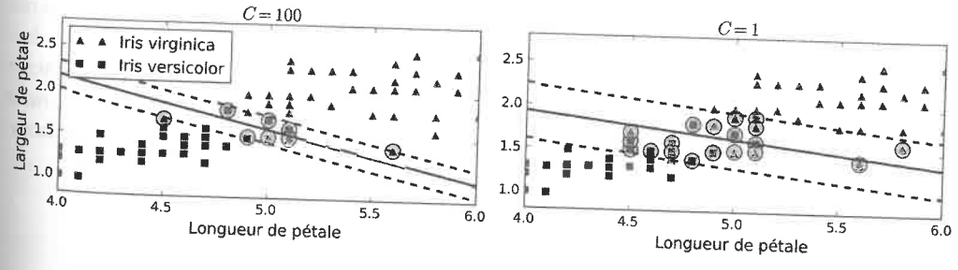


Figure 5.4 – Moins d'empiétements de marge ou marge plus large

Si votre modèle SVM surajuste, vous pouvez tenter de le régulariser en réduisant C.

Le code Scikit-Learn qui suit charge le jeu de données Iris, normalise les variables, puis entraîne un modèle SVM linéaire, en utilisant la classe `LinearSVC` avec C = 0,1 et la fonction de *coût charnière* (en anglais, *hinge loss*) décrite ci-dessous, afin de détecter les fleurs de la variété *Iris virginica*. Le modèle résultant est présenté à droite sur la figure 5.4.

```
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)]  # longueur pétale, largeur pétale
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)
```

Puis vous pouvez, comme d'habitude, utiliser le modèle pour effectuer des prédictions :

```
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```

Contrairement aux classificateurs de régression logistique, les classificateurs SVM ne fournissent pas de probabilités pour chaque classe.

Utiliser la classe `SVC` sous la forme `SVC(kernel="linear", C=1)` constitue une variante, mais elle est beaucoup plus lente, tout particulièrement pour les jeux d'entraînement de grande taille, c'est pourquoi cette solution n'est pas recommandée. Il est aussi possible d'utiliser la classe `SGDClassifier` sous la forme `SGDClassifier(loss="hinge", alpha=1/(m*C))` : ceci effectue une descente de gradient stochastique normale (voir chapitre 4) pour entraîner un classificateur SVM linéaire. Ceci ne converge pas aussi rapidement que la classe `LinearSVC`, mais cela peut être utile pour gérer de grands jeux de données qui ne tiennent pas en mémoire (entraînement hors-mémoire), ou pour gérer des tâches de classification en ligne.

La classe `LinearSVC` régularise le terme constant, c'est pourquoi vous devez d'abord centrer le jeu d'entraînement en soustrayant sa moyenne. Ceci est réalisé automatiquement si vous normalisez les données avec `StandardScaler`. Veillez également à donner explicitement à l'hyperparamètre `loss` la valeur `"hinge"`, car ce n'est pas la valeur par défaut. Enfin, pour améliorer les performances, donnez à l'hyperparamètre `dual` la valeur `False`, sauf s'il y a plus de variables que d'observations d'entraînement (nous parlerons plus loin de la dualité).

5.2 CLASSIFICATION SVM NON LINÉAIRE

Si les classificateurs SVM linéaires sont efficaces et fonctionnent étonnamment bien dans de nombreux cas, de nombreux jeux de données sont fort loin d'être linéairement séparables. Une approche pour gérer ces jeux de données non linéaires consiste à ajouter davantage de variables, par exemple des variables polynomiales (comme nous l'avons fait au chapitre 4) : dans certains cas, le jeu de données qui en résulte est linéairement séparable. Examinez le graphique de gauche de la figure 5.5 : il représente un jeu de données très simple comportant une seule variable x_1. Comme vous pouvez le constater, ce jeu de données n'est pas linéairement séparable. Mais si vous ajoutez une deuxième variable $x_2 = (x_1)^2$, le jeu de données à deux dimensions qui en résulte est linéairement séparable.

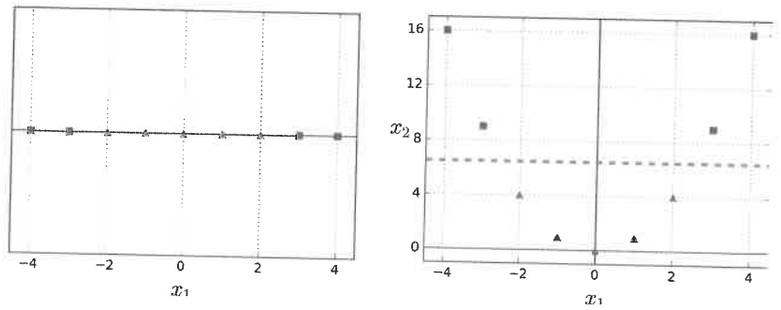


Figure 5.5 – Ajout de variables pour rendre un jeu de données linéairement séparable

Pour implémenter ceci avec Scikit-Learn, vous pouvez créer un `Pipeline` composé d'un transformateur `PolynomialFeatures` (présenté à la section Régression polynomiale du chapitre 4), suivi d'un `StandardScaler` et d'un `LinearSVC`. Testons ceci sur le jeu de données moons (voir figure 5.6).

```
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"))
    ))
polynomial_svm_clf.fit(X, y)
```

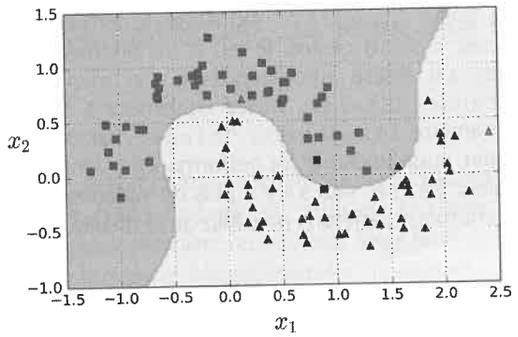


Figure 5.6 – Classificateur SVM linéaire utilisant des variables polynomiales

5.2.1 Noyau polynomial

L'ajout de variables polynomiales est simple à réaliser et peut donner de bons résultats avec toutes sortes d'algorithmes d'apprentissage automatique (et pas seulement les SVM), mais avec un degré polynomial faible on ne peut pas tirer parti de jeux de données très complexes, tandis qu'avec un degré polynomial élevé, on obtient un très grand nombre de variables, ce qui ralentit trop le traitement.

Heureusement, lorsqu'on utilise des SVM, il est possible d'appliquer une technique mathématique quasi miraculeuse appelée l'*astuce du noyau* (expliquée ci-dessous). Elle permet d'obtenir le même résultat que si vous aviez de nombreuses variables polynomiales, même de degré très élevé, sans avoir à les ajouter effectivement. Il n'y a pas donc pas d'explosion combinatoire du nombre des variables, étant donné que vous n'avez pas à les ajouter effectivement. Cette astuce est implémentée par la classe `SVC`. Testons-la sur le jeu de données moons.

```
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)
```

Ce code entraîne un classificateur SVM en utilisant un noyau polynomial de degré 3. Le résultat est présenté à gauche de la figure 5.7. À droite est présenté un autre classificateur SVM utilisant un noyau polynomial de degré 10. Il est clair que si

Une approche courante à la recherche des bonnes valeurs d'hyperparamètres consiste à effectuer une recherche par quadrillage (voir chapitre 2). Il est souvent plus rapide d'effectuer d'abord une recherche avec un quadrillage très grossier, puis une recherche avec un quadrillage plus fin aux alentours des meilleures valeurs trouvées. Pour explorer la bonne partie de l'espace des hyperparamètres, il est utile d'avoir une bonne intuition du rôle exact de chaque hyperparamètre.

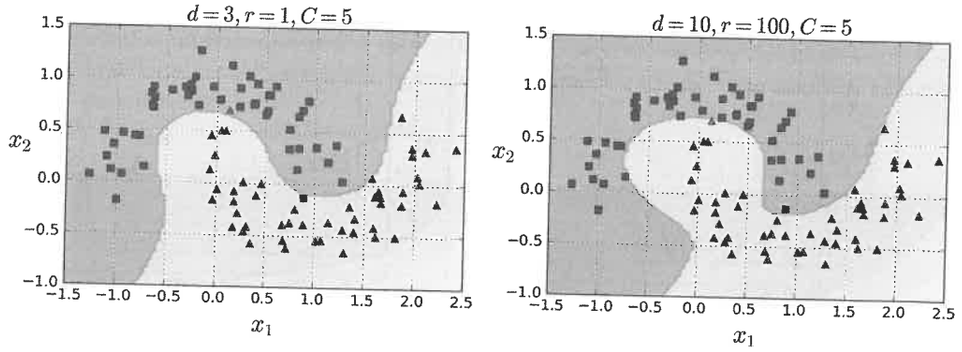


Figure 5.7 – Classificateurs SVM avec noyau polynomial

votre modèle surajuste, vous pouvez envisager de réduire le degré polynomial. Inversement, s'il sous-ajuste, vous pouvez essayer de l'augmenter. L'hyperparamètre `coef0` contrôle l'influence relative sur le modèle des polynômes de degré élevé par rapport aux polynômes de faible degré.

5.2.2 Ajout de variables de similarité

Une autre technique de résolution des problèmes non linéaires consiste à ajouter des variables calculées à l'aide d'une *fonction de similarité* qui mesure la ressemblance entre chaque observation et un *point de repère* (en anglais, *landmark*) particulier. Prenons par exemple le jeu de données unidimensionnel présenté précédemment et ajoutons-lui deux points de repère $x_1 = -2$ et $x_1 = 1$ (voir le graphique de gauche de la figure 5.8). Choisissons ensuite comme fonction de similarité une *fonction de base radiale* (en anglais, Radial Basis Function ou *RBF*) gaussienne où $\gamma = 0{,}3$ (voir équation 5.1).

Équation 5.1 – *RBF gaussienne*

$$\phi_\gamma(\mathbf{x}, \ell) = \exp(-\gamma \|\mathbf{x} - \ell\|^2)$$

C'est une courbe en cloche variant de zéro (très loin du point de repère) jusqu'à 1 (au point de repère). Maintenant, nous sommes prêts à calculer les nouvelles variables. Considérons par exemple l'observation $x_1 = -1$: elle est située à la distance 1 du premier point de repère, et 2 du second point de repère. Par conséquent, les nouvelles variables sont $x_2 = \exp(-0{,}3 \times 1^2) \approx 0{,}74$ et $x_3 = \exp(-0{,}3 \times 2^2) \approx 0{,}30$. Le diagramme de droite de la figure 5.8 présente le jeu de données transformé (en supprimant les variables d'origine). Comme vous pouvez le constater, il est maintenant linéairement séparable.

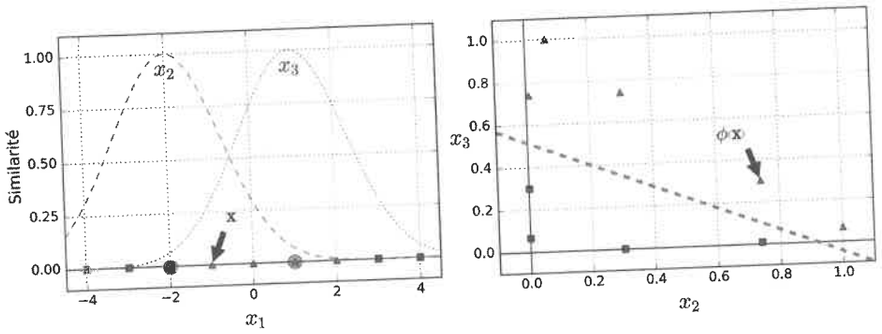


Figure 5.8 – Variables de similarité avec RBF gaussienne

Vous vous demandez peut-être comment sélectionner les points de repère ? La méthode la plus simple consiste à créer un point de repère à l'emplacement de chacune des observations du jeu de données. Ceci crée un très grand nombre de dimensions et accroît par conséquent les chances que le jeu de données transformé soit linéairement séparable. L'inconvénient, c'est qu'un jeu d'entraînement de m observations et de n variables est transformé en un jeu d'entraînement de m observations et de m variables (en supposant que vous abandonniez les variables d'origine). Si votre jeu d'entraînement comporte beaucoup d'observations, vous aboutissez à un nombre tout aussi important de variables.

5.2.3 Noyau radial gaussien

Tout comme la méthode des variables polynomiales, la méthode des variables de similarité peut s'avérer utile avec tout algorithme de Machine Learning, mais il peut être coûteux informatiquement de calculer toutes les variables additionnelles, tout particulièrement lorsque le jeu de données est volumineux. Cependant, l'astuce du noyau fonctionne également dans ce cas : elle permet d'obtenir un résultat semblable à ce qu'on aurait en ajoutant toutes les variables de similarité, mais sans les ajouter effectivement. Essayons le noyau radial gaussien (RBF) avec la classe `SVC`.

```
rbf_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ))
rbf_kernel_svm_clf.fit(X, y)
```

Ce modèle est représenté en bas et à gauche de la figure 5.9. Les autres graphiques correspondent à des modèles entraînés avec différentes valeurs des hyperparamètres `gamma` (γ) et `C`. Accroître `gamma` rend la courbe en cloche plus étroite (voir le diagramme de gauche de la figure 5.8), et diminue par conséquent l'intervalle d'influence de chaque observation : la frontière de décision devient plus irrégulière, se faufilant autour des différentes observations. À l'inverse, réduire la valeur de `gamma` élargit la courbe en cloche et augmente par conséquent la zone d'influence de chaque

observation, ce qui tend à lisser la frontière de décision. Par conséquent, γ fonctionne comme un hyperparamètre de régularisation : si votre modèle surajuste, vous devez le réduire ; et s'il sous-ajuste, vous devez l'augmenter (de manière similaire à l'hyperparamètre `C`).

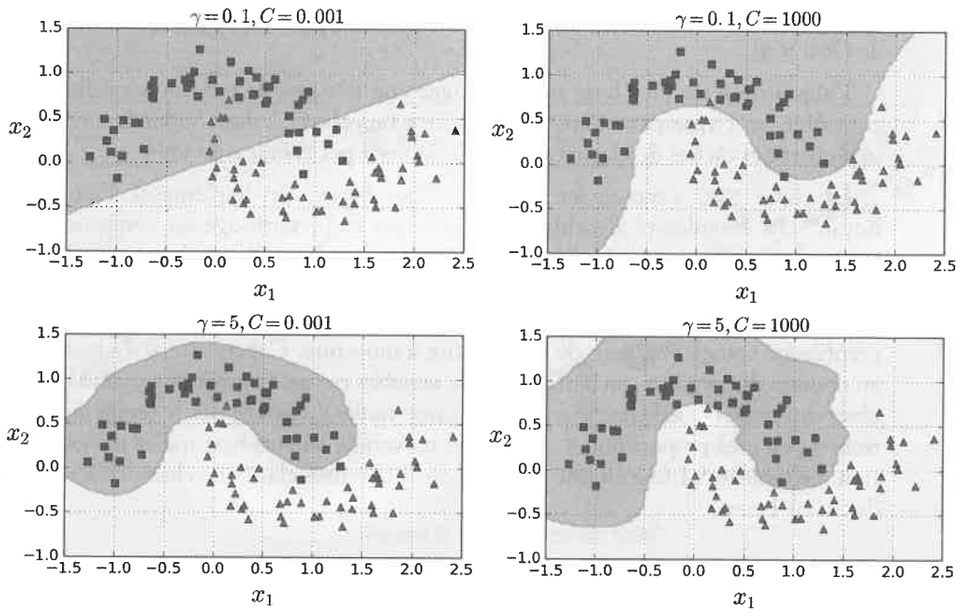


Figure 5.9 – Classificateurs SVM avec noyau radial gaussien

Il existe d'autres noyaux, mais qui sont utilisés plus rarement : par exemple certains noyaux sont adaptés à des structures de données spécifiques : on utilise parfois des *noyaux de chaînes* pour la classification de documents textuels ou de séquences d'ADN (p. ex. un *noyau de sous-chaînes* ou un *noyau basé sur la distance de Levenshtein*).

Comment faire votre choix parmi les nombreux noyaux qui vous sont proposés ? En règle générale, commencez par essayer le noyau linéaire (rappelez-vous que `LinearSVC` est plus rapide que `SVC(kernel="linear")`), en particulier si le jeu d'entraînement comporte beaucoup d'observations ou s'il a beaucoup de variables. Si le jeu d'entraînement n'est pas trop grand, essayez également un noyau radial gaussien (RBF) : il donne de bons résultats dans la plupart des cas. Puis, si vous disposez encore de temps et de ressources informatiques, vous pouvez expérimenter quelques autres noyaux en effectuant une validation croisée et une recherche par quadrillage, en particulier s'il existe des noyaux spécialement adaptés à la structure des données de votre jeu d'entraînement.

5.2.4 Complexité algorithmique

La classe `LinearSVC` s'appuie sur la bibliothèque *liblinear* qui implémente un algorithme optimisé pour les SVM linéaires[63]. Celui-ci ne met pas en œuvre l'astuce du noyau, mais il s'adapte quasi linéairement au nombre d'observations d'entraînement et au nombre de variables : sa complexité, au niveau de l'entraînement, est à peu près de $O(m \times n)$.

L'algorithme est plus long si vous exigez une très grande précision : celle-ci est contrôlée par l'hyperparamètre de tolérance ϵ (appelé `tol` dans Scikit-Learn). Pour la plupart des tâches de classification, la tolérance par défaut convient.

La classe `SVC` s'appuie sur la bibliothèque *libsvm* qui implémente l'astuce du noyau[64]. La complexité algorithmique sur le jeu d'apprentissage est comprise entre $O(m^2 \times n)$ et $O(m^3 \times n)$. Malheureusement, ceci signifie que cet algorithme devient terriblement lent quand le nombre d'observations d'apprentissage est très grand (par exemple, des centaines de milliers). Cet algorithme est parfait pour des jeux d'apprentissage complexes, mais de taille petite à moyenne. Cependant, il s'adapte bien au nombre de variables, en particulier aux *variables creuses* (c'est-à-dire quand chaque observation ne possède que peu de valeurs non nulles) : dans ce cas, le temps de calcul reste à peu près proportionnel au nombre moyen de valeurs non nulles par observation. Le tableau 5.1 fournit une synthèse des différentes classes de classification SVM de Scikit-Learn.

Tableau 5.1 – Comparaison des classes de Scikit-Learn pour la classification SVM

Classe	Complexité algorithmique	Hors-mémoire possible	Normalisation requise	Astuce du noyau
`LinearSVC`	$O(m \times n)$	non	oui	non
`SGDClassifier`	$O(m \times n)$	oui	oui	non
`SVC`	entre $O(m^2 \times n)$ et $O(m^3 \times n)$	non	oui	oui

5.3 RÉGRESSION SVM

Comme nous l'avons mentionné précédemment, l'algorithme SVM est plutôt polyvalent : il permet non seulement la classification linéaire et non linéaire, mais aussi la régression linéaire et non linéaire. Il suffit pour cela de renverser l'objectif : au lieu d'essayer d'ajuster le chemin le plus large possible entre deux classes en limitant les

63. « A Dual Coordinate Descent Method for Large-scale Linear SVM » (une méthode de descente de coordonnées duales pour les SVM linéaires de grande taille), Lin *et al.* (2008) : *http://goo.gl/R635CH*
64. « Sequential Minimal Optimization (SMO) » (optimisation minimale séquentielle), Platt, Microsoft Research (1998), *http://goo.gl/a8HkE3*

empiétements de marge, la régression SVM s'efforce d'ajuster autant d'observations que possible *sur* le chemin tout en limitant les empiétements de marge (c.-à-d. les observations *hors* du chemin). La largeur du chemin est contrôlée par l'hyperparamètre ϵ. La figure 5.10 présente deux modèles de régression SVM linéaire entraînés à partir de quelques données linéaires aléatoires, l'une avec une large marge ($\epsilon = 1{,}5$) l'autre avec une marge étroite ($\epsilon = 0{,}5$).

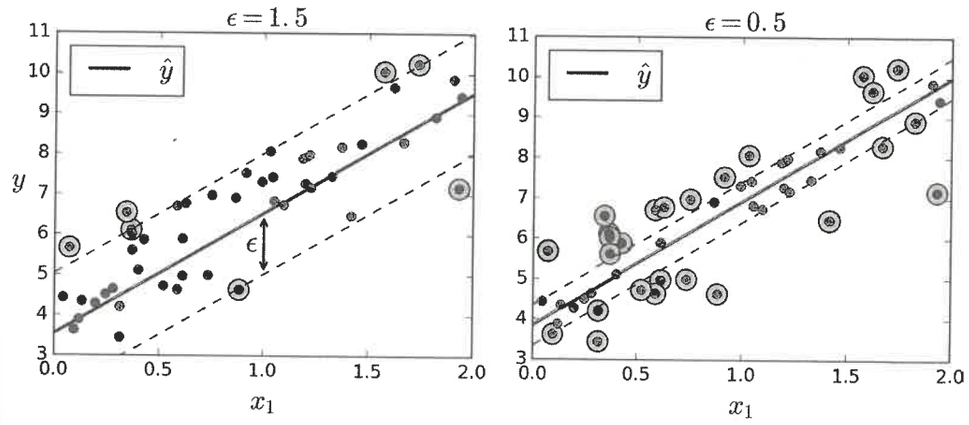


Figure 5.10 – Régression SVM

Ajouter davantage d'observations d'entraînement à l'intérieur de la marge n'affecte pas les prédictions du modèle : celui-ci est dit *insensible à ϵ près*.

Vous pouvez utiliser la classe `LinearSVR` de Scikit-Learn pour effectuer une régression SVM linéaire. Le code suivant produit le modèle représenté à gauche de la figure 5.10 (les données d'entraînement doivent d'abord être normalisées et centrées) :

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

Pour vous attaquer à des tâches de régression non linéaire, vous pouvez utiliser un modèle SVM à noyau. La figure 5.11 présente par exemple une régression SVM sur un jeu d'entraînement quadratique aléatoire, utilisant un noyau polynomial de degré 2. Le graphique de gauche correspond à une faible régularisation (c.-à-d. une valeur `C` élevée), et celui de droite à une régularisation plus importante (c.-à-d. une valeur de `C` plus petite).

Le code qui suit produit le modèle représenté à gauche de la figure 5.11 en utilisant la classe `SVR` de Scikit-Learn (qui implémente l'astuce du noyau). La classe `SVR` est l'équivalent pour la régression de la classe `SVC`, tandis que la classe `LinearSVR` est l'équivalent pour la régression de la classe `LinearSVC`. Le temps de calcul de la classe `LinearSVR` augmente linéairement avec la taille du jeu d'entraînement (comme avec la classe `LinearSVC`), tandis que la classe `SVR`

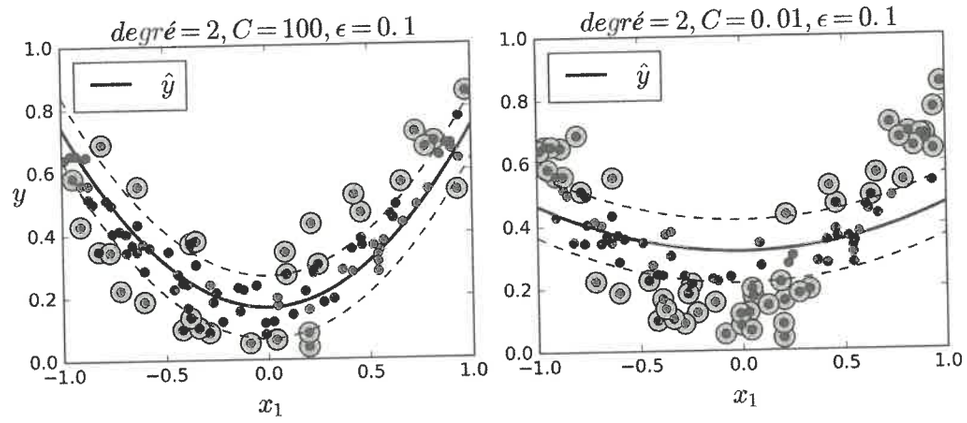


Figure 5.11 – Régression SVM utilisant un noyau polynomial de degré 2

devient bien trop lente lorsque le jeu d'entraînement devient grand (comme avec la classe SVC).

```
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

Les SVM peuvent aussi être utilisés pour détecter les données aberrantes : pour plus de détails, consultez la documentation de Scikit-Learn.

5.4 SOUS LE CAPOT

Cette section explique comment les SVM effectuent des prédictions et comment fonctionnent leurs algorithmes d'entraînement, à commencer par les classificateurs SVM linéaires. Si vous débutez dans le domaine de l'apprentissage automatique, vous pouvez sans problème sauter cette partie et passer directement aux exercices à la fin de ce chapitre. Vous y reviendrez par la suite, lorsque vous souhaiterez mieux comprendre les SVM.

Parlons tout d'abord des notations : au chapitre 4, nous avons adopté la convention consistant à mettre tous les paramètres du modèle dans un même vecteur θ, y compris le terme constant θ_0 et les poids des variables d'entrée θ_1 à θ_n, et d'ajouter une constante $x_0 = 1$ à toutes les observations. Dans ce chapitre, nous allons utiliser une convention différente qui s'avère plus pratique (et plus fréquente) avec les SVM : le terme constant sera appelé b et le vecteur des pondérations des variables sera appelé $\mathbf{w}$. Aucun terme constant ne sera ajouté aux variables.

5.4.1 Fonctions de décision et prédictions

Le modèle de classificateur SVM linéaire prédit la classe d'une nouvelle instance $\mathbf{x}$ en calculant simplement la fonction de décision $\mathbf{w}^T \cdot \mathbf{x} + b = w_1 x_1 + \ldots + w_n x_n + b$: si le résultat est positif, la classe prédite $\hat{y}$ est la classe positive (1), sinon c'est la classe négative (0).

Équation 5.2 – ***Prédiction de classificateur SVM linéaire***

$$\hat{y} = \begin{cases} 0 & \text{si } \mathbf{w}^T \cdot \mathbf{x} + b < 0, \\ 1 & \text{si } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases}$$

La figure 5.12 présente la fonction de décision correspondant au modèle à droite de la figure 5.4 : les données sont représentées dans un même plan, en 2D donc, étant donné que le jeu de données a deux variables (largeur des pétales et longueur des pétales). La frontière de décision est l'ensemble des points pour lesquels la fonction de décision est égale à zéro : c'est l'intersection de deux plans, donc une ligne droite[65] (représentée par un trait plein).

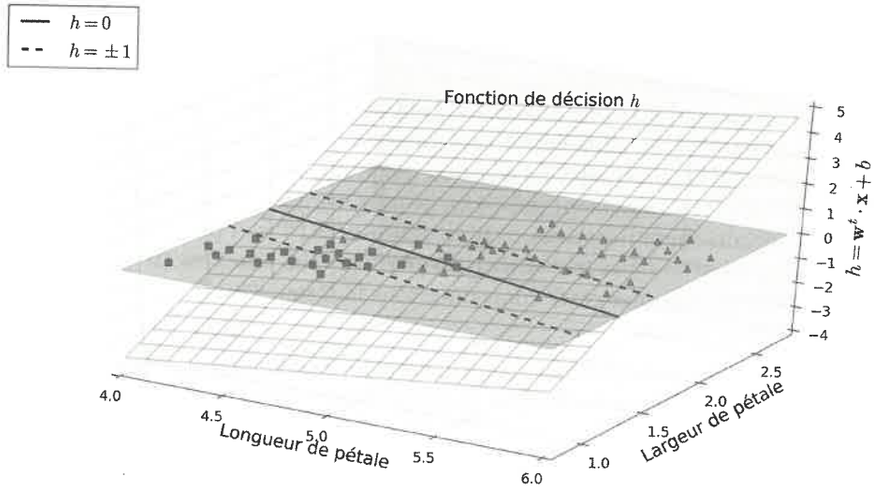


Figure 5.12 – Fonction de décision pour le jeu de données Iris

Les lignes pointillées représentent les points ou la fonction de décision est égale à +1 ou –1 : elles sont parallèles à la frontière de décision et à égale distance de celle-ci, formant une marge autour d'elle. Entraîner un classificateur SVM linéaire consiste à trouver les valeurs de $\mathbf{w}$ et de b rendant cette marge aussi large que possible tout en évitant les empiétements de marge (marge rigide) ou en les limitant (marge souple).

65. Plus généralement, lorsqu'il y a n variables, la fonction de décision est un *hyperplan* à n dimensions, et la frontière de décision est un hyperplan à n–1 dimensions.

5.4.2 Objectif d'entraînement

Considérons la pente de la fonction de décision : elle est égale à la norme du vecteur des pondérations : $\|\mathbf{w}\|$. Si nous divisons cette pente par 2, les points où la fonction de décision est égale à ±1 seront deux fois plus loin de la frontière de décision. En d'autres termes, diviser la pente par 2 revient à multiplier la marge par 2. C'est peut-être plus facile à visualiser en 2D sur la figure 5.13. Plus le vecteur des pondérations $\mathbf{w}$ est petit, plus la marge est grande.

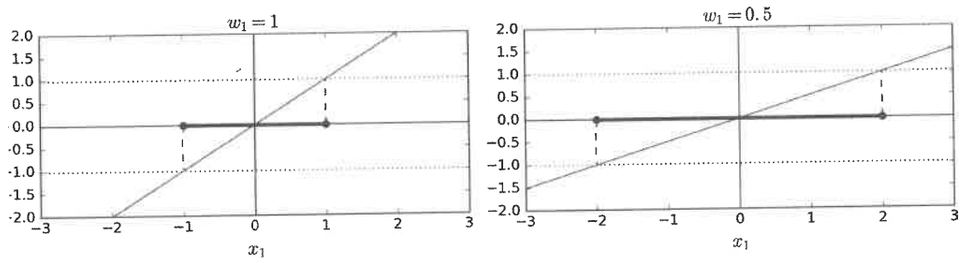


Figure 5.13 – Le vecteur des pondérations le plus petit correspond à la plus large marge

Nous voulons donc minimiser $\|\mathbf{w}\|$ pour obtenir une large marge. Cependant, si nous voulons aussi éviter tout empiètement de marge (marge rigide) alors il nous faut une fonction de décision qui soit supérieure à 1 pour toutes les observations d'entraînement positives, et inférieure à –1 pour toutes les observations d'entraînement négatives. Si nous définissons $t^{(i)} = -1$ pour les observations négatives (si $y^{(i)} = 0$) et $t^{(i)} = 1$ pour les observations positives (si $y^{(i)} = 1$), alors nous pouvons exprimer cette contrainte sous la forme $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$ pour toutes les observations.

Nous pouvons par conséquent exprimer l'objectif d'un classificateur SVM linéaire à marge rigide sous forme d'un problème d'*optimisation sous contraintes* :

Équation 5.3 – ***Objectif d'un classificateur SVM linéaire à marge rigide***

$$\begin{aligned} &\underset{\mathbf{w},b}{\text{minimiser}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} \\ &\text{avec} \quad t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \text{pour } i = 1, 2, \ldots, m \end{aligned}$$

Nous chercherons à minimiser $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{x}$, qui est égal à $\frac{1}{2}\|\mathbf{w}\|^2$, plutôt qu'à minimiser $\|\mathbf{w}\|$: ceci fournira le même résultat (la fonction de transformation étant croissante sur l'intervalle de variation de $\|\mathbf{w}\|$), mais $\frac{1}{2}\|\mathbf{w}\|^2$ possède une dérivée très simple (qui est tout simplement $\mathbf{w}$) alors que $\|\mathbf{w}\|$ n'est pas différentiable en $\mathbf{w} = 0$. Les algorithmes d'optimisation donnent de meilleurs résultats sur les fonctions différentiables.

Pour définir l'objectif correspondant à une marge souple, nous devons introduire une *variable ressort* (en anglais, *slack variable*) $\zeta^{(i)} \geq 0$ pour chaque observation[66] : $\zeta^{(i)}$ mesure de combien la $i^{\text{ième}}$ observation est autorisée à empiéter sur la marge. Nous avons maintenant deux objectifs contradictoires : diminuer autant que possible les variables ressort pour réduire les empiétements de marge, et minimiser $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$ pour accroître la marge. C'est là qu'intervient l'hyperparamètre C : il permet de définir un compromis entre ces deux objectifs. Ceci conduit au problème d'optimisation sous contraintes suivant :

Équation 5.4 – ***Objectif d'un classificateur SVM linéaire à marge souple***

$$\begin{aligned} &\underset{\mathbf{w},b,\zeta}{\text{minimiser}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)} \\ &\text{avec} \quad t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{et} \quad \zeta^{(i)} \geq 0 \quad \text{pour } i = 1, 2, \ldots, m \end{aligned}$$

5.4.3 Programmation quadratique

Les problèmes d'optimisation à marge rigide et à marge souple sont tous les deux des problèmes d'optimisation quadratique avec contrainte linéaire. Les problèmes de ce type sont aussi regroupés sous l'appellation de *programmation quadratique* (en anglais, *quadratic programming*, souvent abrégé en *QP*). On trouve beaucoup de solutions informatiques toutes prêtes pour résoudre ces problèmes de programmation quadratique, mettant en œuvre des techniques sortant du cadre de ce livre[67]. La formulation générale du problème est donnée par l'équation 5.5 :

Équation 5.5 – ***Problème de programmation quadratique***

$$\begin{aligned} &\underset{\mathbf{p}}{\text{Minimiser}} \quad \frac{1}{2}\mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} \quad + \quad \mathbf{f}^T \cdot \mathbf{p} \\ &\text{avec} \quad \mathbf{A} \cdot \mathbf{p} \leq \mathbf{b} \\ &\text{où} \begin{cases} \mathbf{p} & \text{est un vecteur de dimension } n_p \ (n_p = \text{nombre de paramètres}), \\ \mathbf{H} & \text{est une matrice de dimension } n_p \times n_p, \\ \mathbf{f} & \text{est un vecteur de dimension } n_p, \\ \mathbf{A} & \text{est une matrice de dimension } n_c \times n_p \ (n_c = \text{nombre de contraintes}), \\ \mathbf{b} & \text{est un vecteur de dimension } n_c. \end{cases} \end{aligned}$$

Notez que l'expression $\mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$ définit en fait n_c contraintes : $\mathbf{p}^T \cdot \mathbf{a}^{(i)} \leq b^{(i)}$ pour $i = 1, 2, \ldots, n_c$, où $\mathbf{a}^{(i)}$ est le vecteur comportant les éléments de la $i^{\text{ième}}$ ligne de $\mathbf{A}$ et $b^{(i)}$ est le $i^{\text{ième}}$ élément de $\mathbf{b}$.

66. Zêta (ζ) est la 6e lettre de l'alphabet grec.
67. Pour en savoir plus sur la programmation quadratique, vous pouvez commencer par lire « Convex Optimization » de S. Boyd et L. Vandenberghe (*http://goo.gl/FGXuLw*) ou regarder les conférences vidéo de R. Brown : *http://goo.gl/rTo3Af*

Vous pouvez aisément vérifier que si vous définissez les paramètres de programmation quadratique comme ci-dessous, vous obtenez l'objectif d'un classificateur SVM linéaire à marge rigide :

- $n_p = n + 1$, où n est le nombre de variables (le +1 correspond au terme constant),
- $n_c = m$, où m est le nombre d'observations d'entraînement,
- $\mathbf{H}$ est la matrice identité $n_p \times n_p$, à l'exception d'un zéro dans la cellule en haut à gauche (pour ignorer le terme constant),
- $\mathbf{f} = \mathbf{0}$, un vecteur n_p-dimensionnel rempli de 0,
- $\mathbf{b} = \mathbf{1}$, un vecteur n_c-dimensionnel rempli de 1,
- $\mathbf{a}^{(i)} = -t^{(i)}\,\dot{\mathbf{x}}^{(i)}$, où $\dot{\mathbf{x}}^{(i)}$ est égal à $\mathbf{x}^{(i)}$ avec une composante supplémentaire $\dot{\mathbf{x}}_0 = 1$.

Pour entraîner un classificateur SVM linéaire à marge rigide, l'un des moyens consiste à utiliser un des solveurs QP proposés sur le marché, en lui transmettant les paramètres ci-dessus. Le vecteur $\mathbf{p}$ résultant comportera le terme constant $b = p_0$ et les pondérations $w_i = p_i$ pour $i = 1, 2, \ldots, m$. Vous pouvez également utiliser un solveur QP pour résoudre un problème à marge souple (cf. exercices à la fin du chapitre).

Cependant, l'astuce du noyau repose sur un problème d'optimisation sous contraintes différent.

5.4.4 Le problème dual

Étant donné un problème d'optimisation sous contraintes, dénommé *problème primal*, il est possible de l'exprimer sous une forme différente, mais étroitement liée, appelée *problème dual*. La solution du problème dual donne en général un minorant de la solution du problème primal, mais sous certaines conditions les deux problèmes peuvent avoir la même solution. Par chance, le problème SVM remplit ces conditions[68], C'est pourquoi vous pouvez choisir de résoudre le problème primal ou le problème dual, les deux ayant la même solution. L'équation 5.6 présente la forme duale de l'objectif SVM linéaire (si vous souhaitez comprendre comment l'on passe du problème primal au problème dual, reportez-vous à l'annexe C).

Équation 5.6 – *Forme duale de l'objectif SVM linéaire*

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

$$\text{avec} \quad \alpha^{(i)} \geq 0 \quad \text{pour } i = 1, 2, \ldots, m$$

Une fois que vous avez trouvé (à l'aide d'un solveur quadratique) le vecteur $\hat{\boldsymbol{\alpha}}$ qui minimise cette équation, vous pouvez calculer à l'aide des formules 5.7 les $\hat{\mathbf{w}}$ et $\hat{b}$ qui minimisent le problème primal :

68. La fonction d'objectif est convexe, et les contraintes d'inégalité sont des fonctions continûment différentiables et convexes.

Équation 5.7 – *De la solution duale à la solution primale*

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

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

Le problème dual est plus facile à résoudre que le problème primal lorsque le nombre d'observations d'entraînement est plus petit que le nombre de variables. Mais le plus important, c'est qu'il permet l'astuce du noyau, contrairement au problème primal. Mais qu'est-ce donc que cette astuce du noyau ?

5.4.5 SVM à noyau

Supposons que vous vouliez appliquer une transformation polynomiale du second degré à un jeu d'entraînement bidimensionnel (comme le jeu moons par exemple), puis entraîner un classificateur SVM linéaire sur le jeu d'entraînement transformé. L'équation 5.8 présente la fonction polynomiale du second degré ϕ que vous souhaitez appliquer :

Équation 5.8 – *Transformation polynomiale du second degré*

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix}x_1\\x_2\end{pmatrix}\right) = \begin{pmatrix}{x_1}^2\\\sqrt{2}x_1x_2\\{x_2}^2\end{pmatrix}$$

Vous remarquerez que le vecteur transformé a trois dimensions, et non plus deux. Regardons maintenant ce qu'il advient d'un couple de vecteurs bidimensionnels $\mathbf{a}$ et $\mathbf{b}$ si nous leur appliquons cette transformation polynomiale du second degré puis calculons le produit scalaire des vecteurs transformés :

Équation 5.9 – *Astuce du noyau dans le cas d'une transformation polynomiale du second degré*

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

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

Intéressant, n'est-ce pas ? Le produit scalaire des vecteurs transformés est égal au carré du produit scalaire des vecteurs d'origine : $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$.

Et maintenant, voici l'idée clé : si vous appliquez la transformation ϕ à toutes les observations d'entraînement, alors le problème dual (voir équation 5.6) comporte le produit scalaire $\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(j)})$. Mais si ϕ est la transformation polynomiale du second degré définie ci-dessus (voir équation 5.8), alors vous pouvez tout simplement

remplacer ce produit scalaire des vecteurs transformés par $(\mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)})^2$. Par conséquent, vous n'avez pas réellement besoin de transformer les observations d'entraînement : il suffit de remplacer le produit scalaire des variables transformées par le carré du produit scalaire de départ dans l'équation 5.6. Le résultat sera absolument identique à ce que vous obtiendriez en transformant effectivement le jeu d'entraînement puis en appliquant un modèle SVM linéaire, mais cette astuce simplifie grandement l'ensemble de l'algorithme. C'est le principe de ce qu'on appelle l'astuce du noyau.

La fonction $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$ est appelée *noyau polynomial* du second degré. En apprentissage automatique, un *noyau* (en anglais, *kernel*) est une fonction permettant de calculer le produit scalaire $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$ en utilisant uniquement les vecteurs d'origine $\mathbf{a}$ et $\mathbf{b}$, sans avoir à calculer (ni même à connaître) la transformation ϕ. L'équation 5.10 donne la liste des noyaux les plus couramment utilisés.

Équation 5.10 – *Noyaux courants*

$$\begin{aligned}
\text{Linéaire :} \quad & K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b} \\
\text{Polynomial:} \quad & K(\mathbf{a}, \mathbf{b}) = \left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r\right)^d \\
\text{Radial gaussien:} \quad & K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \left\|\mathbf{a} - \mathbf{b}\right\|^2) \\
\text{Sigmoïde :} \quad & K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r\right)
\end{aligned}$$

Théorème de Mercer

Selon le théorème de Mercer, si une fonction $K(\mathbf{a}, \mathbf{b})$ respecte quelques conditions mathématiques appelées *conditions de Mercer* (à savoir, que K doit être continue, symétrique dans ses arguments c'est-à-dire que $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), alors il existe une fonction ϕ projetant $\mathbf{a}$ et $\mathbf{b}$ dans un autre espace (éventuellement de bien plus grande dimension) telle que $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$. Vous pouvez donc utiliser K comme noyau car vous savez que ϕ *existe, même si vous ne connaissez pas cette fonction*. Dans le cas du noyau radial gaussien, on peut démontrer que ϕ projette en fait chaque observation d'entraînement dans un espace de dimension infinie, c'est pourquoi il est heureux que vous n'ayez pas à effectuer réellement la transformation !

Notez que certains noyaux fréquemment utilisés (comme le noyau sigmoïde) ne respectent pas toutes les conditions de Mercer, mais fonctionnent en général fort bien en pratique.

Il nous reste encore un point à régler : l'équation 5.7 montre comment on passe de la solution duale à la solution primale dans le cas d'un classificateur SVM linéaire, mais si vous appliquez l'astuce du noyau vous vous retrouvez avec des équations comportant des $\phi(x^{(i)})$. En fait, $\hat{\mathbf{w}}$ doit avoir le même nombre de dimensions que $\phi(x^{(i)})$, ce qui peut être très grand ou même infini, et vous ne pouvez donc pas le calculer. Mais comment faire des prédictions sans connaître $\hat{\mathbf{w}}$? Heureusement vous pouvez intégrer la formule de calcul de $\hat{\mathbf{w}}$ de l'équation 5.7 dans la fonction de décision pour une nouvelle instance $\mathbf{x}^{(n)}$, et vous obtenez une équation ne comportant que des produits scalaires entre les vecteurs d'entrée. Ainsi, vous pouvez utiliser à nouveau l'astuce du noyau.

Équation 5.11 – *Réalisation de prédictions avec un SVM à noyau*

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

Étant donné que $\alpha^{(i)} \neq 0$ pour les vecteurs de support uniquement, pour effectuer une prédiction il suffit de calculer le produit scalaire du nouveau vecteur d'entrée $\mathbf{x}^{(n)}$ uniquement avec les vecteurs de support, et non pas avec toutes les observations d'entraînement. Bien sûr, vous devez aussi calculer le terme constant $\hat{b}$ en utilisant la même astuce.

Équation 5.12 – *Calcul du terme constant en utilisant l'astuce du noyau*

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

Si vous commencez à avoir mal à la tête, c'est parfaitement normal : c'est un des effets de bord de l'astuce du noyau.

5.4.6 SVM en ligne

Avant de conclure ce chapitre, effectuons un survol rapide des classificateurs SVM en ligne (rappelez-vous que l'apprentissage en ligne correspond à un apprentissage progressif, en général au fur et à mesure de l'arrivée de nouvelles observations).

Pour les classificateurs SVM linéaires, une des méthodes consiste à utiliser une descente de gradient (c.-à-d. à utiliser `SGDClassifier`) pour minimiser la fonction de coût suivante, dérivée du problème primal. Malheureusement celle-ci converge beaucoup plus lentement que les méthodes basées sur la programmation quadratique.

Équation 5.13 – *Fonction de coût d'un classificateur SVM linéaire*

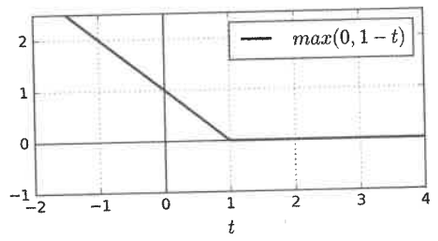
$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} \quad + \quad C \sum_{i=1}^{m} max\left(0, 1 - t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b)\right)$$

Le premier terme de la fonction de coût contraindra le modèle à avoir un petit vecteur de pondération $\mathbf{w}$, ce qui conduira à une large marge. Le second terme correspond au total de tous les empiétements de marge : l'empiètement de marge d'une

observation donnée est égal à 0 si celle-ci se trouve en dehors du chemin et du bon côté, et sinon est proportionnel à la distance au bord correct du chemin. En minimisant ce terme, on minimise les empiétements de marge de ce modèle.

Coût charnière

La fonction *max*(0, 1 – *t*) présentée ci-dessous est appelée fonction de *coût charnière* (en anglais, *hinge loss*). Elle est égale à 0 lorsque $t \geq 1$. Sa dérivée (pente) est égale à –1 si $t < 1$ et 0 si $t > 1$. Elle n'est pas différentiable en $t = 1$, mais, tout comme pour la régression lasso (voir chapitre 4), vous pouvez malgré tout effectuer une descente de gradient en utilisant n'importe quelle sous-dérivée en $t = 1$ (c.-à-d. n'importe quelle valeur comprise entre –1 et 0).



Il est aussi possible d'implémenter des SVM à noyau en ligne, en utilisant par exemple une méthode incrémentale et décrémentale[69], ou un classificateur à noyau rapide[70]. Toutefois, ces algorithmes sont implémentés en Matlab et C++. Pour les problèmes non linéaires à grande échelle, vous pourrez plutôt envisager d'utiliser des réseaux neuronaux (voir l'ouvrage *Deep Learning avec TensorFlow*, A. Géron, Dunod, 2017).

5.5 EXERCICES

1. Quel est le principe fondamental des machines à vecteurs de support ? Sous quel autre nom les connaît-on ?
2. Qu'est-ce qu'un vecteur de support ?
3. Pourquoi est-ce important de normaliser les données d'entrée lorsqu'on utilise des SVM ?
4. Un classificateur SVM peut-il fournir un indice de confiance lorsqu'il classe une observation ? Et peut-il fournir une probabilité ?
5. Si votre jeu d'entraînement possède des millions d'observations et

69. « Incremental and Decremental SVM Learning », Cauwenberghs, Poggio (2001), NIPS : *http://goo.gl/JEqVui*
70. « Fast Kernel Classifiers with Online and Active Learning », Bordes, Ertekin, Weston, Bottou (2005), Journal of Machine Learning Research : *http://goo.gl/42K9kG*

des centaines de variables devez-vous utiliser la forme primale du problème SVM ou sa forme duale pour entraîner le modèle ?

6. Supposons que vous ayez entraîné un classificateur SVM avec un noyau à base radiale. Il semble sous-ajuster le jeu d'entraînement : devez-vous augmenter ou diminuer γ (`gamma`) ? Même question pour `C`.
7. Comment devez-vous définir les paramètres de programmation quadratique (**H**, **f**, **A** et **b**) pour résoudre un problème de classification SVM linéaire à marge souple en utilisant un solveur QP proposé sur le marché ?
8. Entraînez un `LinearSVC` sur un jeu de données linéairement séparable. Puis entraînez un `SVC` et un `SGDClassifier` sur le même jeu de données. Voyez si vous pouvez leur faire produire à peu près le même modèle.
9. Entraînez un classificateur SVM sur le jeu de données MNIST. Les classificateurs SVM étant des classificateurs binaires, vous devrez utiliser une stratégie un-contre-tous pour classer les 10 chiffres. Pour accélérer le processus, vous pouvez essayer de régler les hyperparamètres en utilisant de petits jeux de validation. Quelle précision pouvez-vous atteindre ?
10. Entraînez un régresseur SVM sur le jeu de données des prix immobiliers en Californie.

Les solutions de ces exercices sont données à l'annexe A.