Institut National des Postes et Télécommunications

Filière: Data Engineer 2

Machine Learning TP1: Supervised Learning

Pr. Tarik Fissaa

Ce TP propose un ensemble d'exercices pour différentes tâches avec Scikit-Learn.

Remarques :

- Il peut y avoir plus d'une manière différente de répondre à une question ou d'effectuer un exercice.
- Un squelette de code a été implémenté pour vous.
- Les différentes tâches seront détaillées par des commentaires ou du texte. Les emplacements pour mettre votre propre code sont définis par ### (ne supprimez rien d'autre que ###).

Pour plus d'informations et de ressources, il est conseillé de consulter la documentation Scikit-Learn.

Et si vous êtes bloqué, essayez de rechercher une question au format suivant : "comment faire XYZ avec Scikit-Learn", où XYZ est la fonction que vous souhaitez exploiter de Scikit-Learn.

Puisque nous travaillerons avec des données, nous importerons en plus de **Scikit-Learn**, **Matplotlib**, **NumPy** et **pandas**.

Let's get started.

In [ ]:
```
# Setup matplotlib to plot inline (within the notebook)
###

# Import the pyplot module of Matplotlib as plt
###

# Import pandas under the abbreviation 'pd'
###

# Import NumPy under the abbreviation 'np'
###
```

Workflow de classification Scikit-Learn de bout en bout

Commençons par un workflow Scikit-Learn de bout en bout.

Plus précisément, nous allons :

1- Préparez un jeu de données

2- Préparer un modèle d'apprentissage automatique pour faire des prédictions

3- Ajuster le modèle aux données et faire une prédiction

4- Évaluer les prédictions du modèle

Nous commencerons par *heart-disease.csv*, un ‘’Dataset’’ contenant des données anonymes sur les patients et indiquant s'ils souffrent ou non d'une maladie cardiaque.

1. **Préparer les données :**

In [ ]:
```
# Import the heart disease dataset and save it to a variable
# using pandas and read_csv()

heart_disease = ###

# Check the first 5 rows of the data
###
```

Notre objectif ici est de créer un modèle d'apprentissage automatique sur toutes les colonnes à l'exception de *target* pour prédire *target*.

En substance, la colonne *target* est notre variable cible (également appelée *y* ou label) et le reste des autres colonnes sont nos variables indépendantes (également appelées données ou *X*).

Comme notre variable cible est une catégorie (maladie cardiaque ou non), c’est quoi le type de problème de ML qu’on doit résoudre ?

Sachant cela, créons *X* et *y* en fractionnant un dataframe.

In [ ]:
```
# Create X (all columns except target)
X = ###

# Create y (only the target column)
y = ###
```

Maintenant que nous avons divisé nos données en *X* et *y*, nous utiliserons **Scikit-Learn** pour les diviser en ensembles d'entraînement et de test (training and test).

In [ ]:
```
# Import train_test_split from sklearn's model_selection module
###

# Use train_test_split to split X & y into training and test sets
X_train, X_test, y_train, y_test = ###
```

In [ ]:
```
# View the different shapes of the training and test datasets
###
```

Étant donné que nos données sont maintenant dans des ensembles d'entraînement et de test, nous allons créer un modèle d'apprentissage automatique pour adapter les modèles dans les données d'entraînement, puis effectuer des prédictions sur les données de test.

Pour déterminer quel modèle d'apprentissage automatique nous devrions utiliser, vous pouvez vous référer à « Scikit-Learn's machine learning map » .

Après avoir suivi la carte, vous décidez d'utiliser le RandomForestClassifier.

2. Préparer un modèle d'apprentissage automatique :

```
In [ ]: # Import the RandomForestClassifier from sklearn's ensemble module
        ###

        # Instantiate an instance of RandomForestClassifier as clf
        clf =
```

Maintenant que vous avez une instance RandomForestClassifier, adaptons-la aux données d'entraînement.

Une fois qu'il est adapté, nous ferons des prédictions sur les données de test.

3. Ajuster le modèle et faire des prédictions :

```
In [ ]: # Fit the RandomForestClassifier to the training data
        clf.fit(###, ###)
```

```
In [ ]: # Use the fitted model to make predictions on the test data and
        # save the predictions to a variable called y_preds
        y_preds = clf.predict(###)
```

4. Évaluer les prédictions du modèle :

Évaluer les prédictions est très important. Vérifions comment notre modèle en appelant la méthode *score()* et en lui passant les données d'entraînement *(X_train, y_train)* et de test *(X_test, y_test)*.

```
In [ ]: # Evaluate the fitted model on the training set using the score() function
        ###
```

```
In [ ]: # Evaluate the fitted model on the test set using the score() function
        ###
```

- Comment le modèle a performé ?
- Quelle métrique la fonction score() renvoie-t-elle pour les classificateurs ?
- Votre modèle a-t-il mieux fonctionné avec l'ensemble de données d'entraînement ou l'ensemble de données de test ?

Expérimenter différents modèles de classification

Nous avons maintenant rapidement couvert le workflow Scikit-Learn de bout en bout et comme l'expérimentation est une grande partie de l'apprentissage automatique, nous allons maintenant essayer une série de différents modèles d'apprentissage automatique et voir lequel obtient les meilleurs résultats sur notre ensemble de données.

En parcourant la fiche de Scikit-Learn, nous voyons qu'il existe un certain nombre de modèles de classification différents que nous pouvons essayer (différents modèles sont dans les cases vertes).

Pour cet exercice, les modèles que nous allons essayer de comparer sont :

- LinearSVC
- KNeighborsClassifier (K-Nearest Neighbors or KNN)
- SVC (support vector classifier, une forme de support vector machine)
- LogisticRegression (malgré le nom, c'est en fait un classificateur)

- RandomForestClassifier (une méthode d'ensemble)

Nous suivrons le même workflow que nous avons utilisé ci-dessus (sauf cette fois pour plusieurs modèles):

1- Importer un modèle d'apprentissage automatique
2- Préparez-le
3- Ajustez-le aux données et faites des prédictions
4- Évaluer le modèle ajusté

Remarque : puisque nous avons déjà les données prêtes, nous pouvons les réutiliser dans cette section.

In [1]:
```
# Import LinearSVC from sklearn's svm module
###

# Import KNeighborsClassifier from sklearn's neighbors module
###

# Import SVC from sklearn's svm module
###

# Import LogisticRegression from sklearn's linear_model module
###

# Note: we don't have to import RandomForestClassifier, since we already have
```

Grâce à la cohérence de la conception de l'API de Scikit-Learn, nous pouvons utiliser pratiquement le même code pour ajuster, évaluer et faire des prédictions avec chacun de nos modèles.

Pour voir quel modèle fonctionne le mieux, nous allons procéder comme suit :

1- Instancier chaque modèle dans un dictionnaire
2- Créer un dictionnaire de résultats vide
3- Ajustez chaque modèle sur les données d'entraînement
4- Evaluer chaque modèle sur les données de test
5- Vérifiez les résultats

Si vous vous demandez ce que signifie instancier chaque modèle dans un dictionnaire, consultez l'exemple ci-dessous.

In [ ]:
```
# EXAMPLE: Instantiating a RandomForestClassifier() in a dictionary
example_dict = {"RandomForestClassifier": RandomForestClassifier()}

# Create a dictionary called models which contains all of the classification models we've imported
# Make sure the dictionary is in the same format as example_dict
# The models dictionary should contain 5 models
models = {"LinearSVC": ###,
          "KNN": ###,
          "SVC": ###,
          "LogisticRegression": ###,
          "RandomForestClassifier": ###}

# Create an empty dictionary called results
results = ###
```

Étant donné que chaque modèle que nous utilisons a les mêmes fonctions *fit ()* et *score ()*, nous pouvons parcourir notre dictionnaire de modèles et appeler *fit ()* sur les données d'entraînement, puis appeler *score ()* avec les données de test.

In [ ]:
```
# EXAMPLE: Looping through example_dict fitting and scoring the model
example_results = {}
for model_name, model in example_dict.items():
    model.fit(X_train, y_train)
    example_results[model_name] = model.score(X_test, y_test)

# EXAMPLE: View the results
example_results
```

In [ ]:
```
# Loop through the models dictionary items, fitting the model on the training data
# and appending the model name and model score on the test data to the results dictionary
for model_name, model in ###:
    model.fit(###)
    results[model_name] = model.score(###)

# View the results
results
```

- Quel modèle a le mieux fonctionné ?
- Est-ce que les résultats changent à chaque fois que vous exécutez la cellule ?
- Si ça change, pourquoi à votre avis?

En raison du caractère aléatoire de la façon dont chaque modèle trouve des modèles dans les données, vous remarquerez peut-être des résultats différents à chaque fois.

Sans définir manuellement l'état aléatoire à l'aide du paramètre random_state de certains modèles ou en utilisant NumPy random seed, chaque fois que vous exécutez la cellule, vous obtiendrez des résultats légèrement différents.

Voyons cela en effet en exécutant le même code que la cellule ci-dessus, sauf cette fois en définissant NumPy random seed equal to 42.

Rendons nos résultats un peu plus visuels.

In [ ]:
```
# Create a pandas dataframe with the data as the values of the results dictionary,
# the index as the keys of the results dictionary and a single column called accuracy.
# Be sure to save the dataframe to a variable.
results_df = pd.DataFrame(results.###(),
                          results.###(),
                          columns=[####])

# Create a bar plot of the results dataframe using plot.bar()
###
```

L'utilisation de np.random.seed (42) permet au modèle de régression logistique de fonctionner le mieux (au moins sur mon ordinateur).

Affinons ses hyperparamètres et voyons si nous pouvons l'améliorer.

Hyperparameter Tuning

N'oubliez pas que si vous essayez de régler des hyperparamètres de modèles de machine learning et que vous ne savez pas par où commencer, vous pouvez toujours rechercher quelque chose comme "Réglage des hyperparamètres MODEL_NAME".

Dans le cas de LogisticRegression, vous pourriez rencontrer des articles, tels que Hyperparameter Tuning Using Grid Search de Chris Albon.

L'article utilise GridSearchCV mais nous allons utiliser RandomizedSearchCV.

Les différents hyperparamètres à rechercher ont été configurés pour vous dans log_reg_grid mais n'hésitez pas à les modifier.

In [ ]:
```
# Different LogisticRegression hyperparameters
log_reg_grid = {"C": np.logspace(-4, 4, 20),
                "solver": ["liblinear"]}
```

Puisque nous avons un ensemble d'hyperparamètres, nous pouvons importer RandomizedSearchCV, lui transmettre notre dictionnaire d'hyperparamètres et le laisser rechercher la meilleure combinaison.

In [ ]:
```
# Setup np random seed of 42
np.random.seed(###)

# Import RandomizedSearchCV from sklearn's model_selection module


# Setup an instance of RandomizedSearchCV with a LogisticRegression() estimator,
# our log_reg_grid as the param_distributions, a cv of 5 and n_iter of 5.
rs_log_reg = RandomizedSearchCV(estimator=###,
                                param_distributions=###,
                                cv=###,
                                n_iter=###,
                                verbose=###)

# Fit the instance of RandomizedSearchCV
###
```

Une fois que RandomizedSearchCV a terminé, nous pouvons trouver les meilleurs hyperparmètres trouvés en utilisant les attributs best_params_.

In [ ]:
```
# Find the best parameters of the RandomizedSearchCV instance using the best_params_ attribute
###
```

In [ ]:
```
# Score the instance of RandomizedSearchCV using the test data
###
```

Après le réglage des hyperparamètres, le score des modèles s'est-il amélioré ? Que pourriez-vous essayer d'autre pour l'améliorer ? Existe-t-il d'autres méthodes de réglage des hyperparamètres que vous pouvez trouver pour LogisticRegression ?

Évaluation du modèle de classificateur

Nous avons essayé de trouver les meilleurs hyperparamètres sur notre modèle en utilisant RandomizedSearchCV et jusqu'à présent, nous n'avons évalué notre modèle qu'en utilisant la fonction *score ()* qui renvoie la précision.

Mais en ce qui concerne la classification, vous voudrez probablement utiliser quelques autres métriques d'évaluation, notamment :

- **Confusion matrix** - Compares the predicted values with the true values in a tabular way, if 100% correct, all values in the matrix will be top left to bottom right (diagnol line).
- **Cross-validation** - Splits your dataset into multiple parts and train and tests your model on each part and evaluates performance as an average.
- **Precision** - Proportion of true positives over total number of samples. Higher precision leads to less false positives.
- **Recall** - Proportion of true positives over total number of true positives and false positives. Higher recall leads to less false negatives.
- **F1 score** - Combines precision and recall into one metric. 1 is best, 0 is worst.
- **Classification report** - Sklearn has a built-in function called `classification_report()` which returns some of the main classification metrics such as precision, recall and f1-score.
- **ROC Curve** - Receiver Operating Characterisitc is a plot of true positive rate versus false positive rate.
- **Area Under Curve (AUC)** - The area underneath the ROC curve. A perfect model achieves a score of 1.0.

Avant d'y arriver, nous allons instancier une nouvelle instance de notre modèle en utilisant les meilleurs hyperparamètres trouvés par RandomizedSearchCV.

In [ ]:
```
# Instantiate a LogisticRegression classifier using the best hyperparameters from RandomizedSearchCV
clf = LogisticRegression(###)

# Fit the new instance of LogisticRegression with the best hyperparameters on the training data
###
```

Il s'agit maintenant d'importer les méthodes Scikit-Learn relatives pour chacune des métriques d'évaluation de classification que nous recherchons.

In [ ]:
```
# Import confusion_matrix and classification_report from sklearn's metrics module
###

# Import precision_score, recall_score and f1_score from sklearn's metrics module
###

# Import plot_roc_curve from sklearn's metrics module
###
```

Faisons quelques prédictions sur les données de test en utilisant notre dernier modèle et sauvegardons-les dans y_preds.

In [ ]:
```
# Make predictions on test data and save them
###
```

Il est temps d'utiliser les prédictions que notre modèle a trouvé pour l'évaluer

In [ ]:
```
# Create a confusion matrix using the confusion_matrix function
###
```

Challenge : La fonction confusion_matrix intégrée dans Scikit-Learn produit quelque chose de pas trop visuel, comment pourriez-vous rendre votre matrice de confusion plus visuelle ?

Vous voudrez peut-être rechercher quelque chose comme "comment tracer une matrice de confusion". Remarque : il peut y avoir plus d'une façon de procéder.

In [ ]:
```
# Create a more visual confusion matrix
###
```

Que diriez-vous d'un rapport de classification ?

In [ ]:
```
# Create a classification report using the classification_report function
###
```

Défi : notez ce que sont chacune des colonnes de ce rapport de classification.

- Précision - Indique la proportion d'identifications positives (classe 1 prédite par le modèle) qui étaient réellement correctes. Un modèle qui ne produit aucun faux positif a une précision de 1,0.
- Rappel - Indique la proportion de positifs réels qui ont été correctement classés. Un modèle qui ne produit aucun faux négatif a un rappel de 1,0.
- Score F1 - Une combinaison de précision et de rappel. Un modèle parfait obtient un score F1 de 1,0.
- Support - Le nombre d'échantillons sur lequel chaque métrique a été calculée.
- Précision - La précision du modèle sous forme décimale. La précision parfaite est égale à 1,0.
- Macro moyenne - Abréviation de macro moyenne, la précision moyenne, le rappel et le score F1 entre les classes. La macro moyenne ne classe pas le déséquilibre en effort, donc si vous avez des déséquilibres de classe, faites attention à cette métrique.
- Moyenne pondérée - Abréviation de moyenne pondérée, précision moyenne pondérée, rappel et score F1 entre les classes. Pondéré signifie que chaque métrique est calculée par rapport au nombre d'échantillons dans chaque classe. Cette métrique favorisera la classe majoritaire (par exemple, donnera une valeur élevée lorsqu'une classe surpassera une autre en raison du plus grand nombre d'échantillons).

Le rapport de classification nous donne une plage de valeurs pour la précision, le rappel et le score F1, le temps de trouver ces métriques à l'aide des fonctions Scikit-Learn.

In [ ]:
```
# Find the precision score of the model using precision_score()
###
```

In [ ]:
```
# Find the recall score
###
```

In [ ]:
```
# Find the F1 score
###
```

Matrice de confusion: terminé. Rapport de classification: terminé.

Courbe ROC (caractéristique de l'opérateur du récepteur) et score AUC (aire sous la courbe): pas encore.

Corrigeons ça.

Si vous n'êtes pas familier avec ce qu'est une courbe ROC, c'est votre premier défi, de lire ce que c'est.

En une phrase, une courbe ROC est un tracé du taux de vrais positifs par rapport au taux de faux positifs.

Et le score AUC est la zone derrière la courbe ROC.

Scikit-Learn fournit une fonction pratique pour créer les deux, appelée plot_roc_curve ().

In [ ]:
```
# Plot a ROC curve using our current machine learning model using plot_roc_curve
###
```

Magnifique ! Nous sommes allés bien au-delà de la précision avec une pléthore de paramètres d'évaluation de classification supplémentaires.

Si vous n'êtes pas sûr de l'un de ces éléments, ne vous inquiétez pas, ils peuvent prendre un certain temps à comprendre. Cela pourrait être une extension facultative, en lisant une métrique de classification dont vous n'êtes pas sûr.

La chose à noter ici est que toutes ces métriques ont été calculées à l'aide d'un seul ensemble d'apprentissage et d'un seul ensemble de test. Bien que ce soit correct, un moyen plus robuste consiste à les calculer à l'aide de la validation croisée.

Nous pouvons calculer diverses métriques d'évaluation en utilisant la validation croisée en utilisant la fonction cross_val_score () de Scikit-Learn avec le paramètre de score.

In [ ]:
```
# Import cross_val_score from sklearn's model_selection module
###
```

In [ ]:
```
# EXAMPLE: By default cross_val_score returns 5 values (cv=5).
cross_val_score(clf,
                X,
                y,
                scoring="accuracy",
                cv=5)
```

In [ ]:
```
# EXAMPLE: Taking the mean of the returned values from cross_val_score
# gives a cross-validated version of the scoring metric.
cross_val_acc = np.mean(cross_val_score(clf,
                                        X,
                                        y,
                                        scoring="accuracy",
                                        cv=5))

cross_val_acc
```

Dans les exemples, la précision de la validation croisée est trouvée en prenant la moyenne du tableau retourné par cross_val_score ().

Il est maintenant temps de trouver la même chose pour la précision, le rappel et le score F1.

In [ ]:
```
# Find the cross-validated precision
###
```

In [ ]:
```
# Find the cross-validated recall
###
```

In [ ]:
```
# Find the cross-validated F1 score
###
```

Astuce: Exporting and importing a trained model

Une fois que vous avez formé un modèle, vous souhaiterez peut-être l'exporter et l'enregistrer dans un fichier afin de pouvoir le partager ou l'utiliser ailleurs.

Une méthode d'exportation et d'importation de modèles consiste à utiliser la bibliothèque joblib.

Dans Scikit-Learn, l'exportation et l'importation d'un modèle entraîné s'appelle la persistance du modèle.

In [ ]:
```
# Import the dump and load functions from the joblib library
###
```

In [ ]:
```
# Use the dump function to export the trained model to file
###
```

In [ ]:
```
# Use the load function to import the trained model you just exported
# Save it to a different variable name to the origial trained model
###

# Evaluate the loaded trained model on the test data
###
```

Que remarquez-vous à propos des résultats du modèle entraîné chargé par rapport aux résultats du modèle d'origine (pré-exportés).

Pratique de la régression en Scikit-Learn

Pour les prochains exercices, nous allons travailler sur un problème de régression, c'est-à-dire en utilisant certaines données pour prédire un nombre.

Notre ensemble de données est un tableau des ventes de voitures, contenant différentes caractéristiques de voitures ainsi qu'un prix de vente.

Nous utiliserons les modèles d'apprentissage automatique de régression intégrés de Scikit-Learn pour essayer d'apprendre les modèles dans les caractéristiques des voitures et leurs prix sur un certain groupe de l'ensemble de données avant d'essayer de prédire le prix de vente d'un groupe de voitures que le modèle n'a jamais.

Pour commencer, nous allons importer les données dans un pandas DataFrame, vérifier quelques détails à ce sujet et essayer de créer un modèle dès que possible.

In [ ]:
```
# Read in the car sales data
car_sales = pd.read_csv("https://raw.githubusercontent.com/mrdbourke/zero-to-mastery-ml/master/data/car-sales-extended-missing-data.csv")

# View the first 5 rows of the car sales data
###
```

In [ ]:
```
# Get information about the car sales DataFrame
###
```

En regardant la sortie de info(),

- Combien de lignes y a-t-il au total ?
- Quels types de données se trouvent dans chaque colonne ?
- Combien de valeurs manquantes y a-t-il dans chaque colonne ?

In [ ]:
```
# Find number of missing values in each column
###
```

In [ ]:
```
# Find the datatypes of each column of car_sales
###
```

Connaissant ces informations, que se passerait-il si nous essayions de modéliser nos données telles quelles ?

Voyons voir.

In [ ]:
```
# EXAMPLE: This doesn't work because our car_sales data isn't all numerical
from sklearn.ensemble import RandomForestRegressor
car_sales_X, car_sales_y = car_sales.drop("Price", axis=1), car_sales.Price
rf_regressor = RandomForestRegressor().fit(car_sales_X, car_sales_y)
```

Comme nous le voyons, la cellule ci-dessus se brise car nos données contiennent des valeurs non numériques ainsi que des données manquantes.

Pour prendre en charge certaines des données manquantes, nous supprimerons les lignes qui n'ont pas d'étiquettes (toutes les lignes avec des valeurs manquantes dans la colonne Prix).

In [ ]:
```
# Remove rows with no labels (NaN's in the Price column)
###
```

Construire un pipeline

Étant donné que nos données car_sales ont des valeurs numériques manquantes et que les données ne sont pas toutes numériques, nous devrons résoudre ces problèmes avant de pouvoir y adapter un modèle d'apprentissage automatique.

Il existe des moyens de le faire avec pandas, mais puisque nous pratiquons Scikit-Learn, nous verrons comment nous pourrions le faire avec la classe Pipeline.

Étant donné que nous modifions les colonnes de notre dataframe (remplissage des valeurs manquantes, conversion de données non numériques en nombres), nous aurons également besoin des classes ColumnTransformer, SimpleImputer et OneHotEncoder.

Enfin, comme nous devrons diviser nos données en ensembles d'entraînement et de test, nous importerons également train_test_split.

In [ ]:
```
# Import Pipeline from sklearn's pipeline module
###

# Import ColumnTransformer from sklearn's compose module
###

# Import SimpleImputer from sklearn's impute module
###

# Import OneHotEncoder from sklearn's preprocessing module
###

# Import train_test_split from sklearn's model_selection module
###
```

Nous disposons maintenant des outils nécessaires pour créer notre pipeline de prétraitement qui remplit les valeurs manquantes tout en transformant toutes les données non numériques en nombres.

Commençons par les fonctionnalités catégoriques.

In [ ]:
```
# Define different categorical features
categorical_features = ["Make", "Colour"]

# Create categorical transformer Pipeline
categorical_transformer = Pipeline(steps=[
    # Set SimpleImputer strategy to "constant" and fill value to "missing"
    ("imputer", SimpleImputer(strategy=###, fill_value=###)),
    # Set OneHotEncoder to ignore the unknowns
    ("onehot", OneHotEncoder(handle_unknown=###))])
```

Il serait également prudent de traiter les portes comme une caractéristique catégorique, mais comme nous savons que la grande majorité des voitures ont 4 portes, nous imputerons les valeurs de portes manquantes à 4.

In [ ]:
```
# Define Doors features
door_feature = ["Doors"]

# Create Doors transformer Pipeline
door_transformer = Pipeline(steps=[
    # Set SimpleImputer strategy to "constant" and fill value to 4
    ("imputer", SimpleImputer(strategy=###, fill_value=###))])
```

Passons maintenant aux caractéristiques numériques. Dans ce cas, la seule caractéristique numérique est la colonne Odomètre (KM). Remplissons ses valeurs manquantes avec la médiane.

In [ ]:
```
# Define numeric features (only the Odometer (KM) column)
numeric_features = ["Odometer (KM)"]

# Crearte numeric transformer Pipeline
numeric_transformer = ###(steps=[
    # Set SimpleImputer strategy to fill missing values with the "Median"
    ("imputer", ###(strategy=###))])
```

Il est temps de mettre tous nos pipelines de transformateurs individuels dans une seule instance de ColumnTransformer.

In [ ]:
```
# Setup preprocessing steps (fill missing values, then convert to numbers)
preprocessor = ColumnTransformer(
    transformers=[
        # Use the categorical_transformer to transform the categorical_features
        ("cat", categorical_transformer, ###),
        # Use the door_transformer to transform the door_feature
        ("door", ###, door_feature),
        # Use the numeric_transformer to transform the numeric_features
        ("num", ###, ###)])
```

Maintenant, notre preprocessing est prêt, il est temps d'importer des modèles de régression pour les essayer.

En comparant nos données à la carte d'apprentissage automatique Scikit-Learn, nous pouvons voir qu'il existe une poignée de modèles de régression différents que nous pouvons essayer.

- RidgeRegression
- SVR(kernel="linear") - short for Support Vector Regressor, a form of support vector machine.
- SVR(kernel="rbf") - short for Support Vector Regressor, a form of support vector machine.
- RandomForestRegressor - the regression version of RandomForestClassifier.

In [ ]:
```
# Import Ridge from sklearn's linear_model module


# Import SVR from sklearn's svm module


# Import RandomForestRegressor from sklearn's ensemble module
```

Encore une fois, grâce à la conception de la bibliothèque Scikit-Learn, nous sommes en mesure d'utiliser un code très similaire pour chacun de ces modèles.
Pour les tester tous, nous allons créer un dictionnaire de modèles de régression et un dictionnaire vide pour les résultats des modèles de régression.

In [ ]:
```
# Create dictionary of model instances, there should be 4 total key, value pairs
# in the form {"model_name": model_instance}.
# Don't forget there's two versions of SVR, one with a "linear" kernel and the
# other with kernel set to "rbf".
regression_models = {"Ridge": ###,
                     "SVR_linear": ###,
                     "SVR_rbf": ###,
                     "RandomForestRegressor": ###}

# Create an empty dictionary for the regression results
regression_results = ###
```

Notre dictionnaire de modèles de régression est préparé ainsi qu'un dictionnaire vide auquel ajouter les résultats, le temps de diviser les données en X (variables de fonctionnalité) et y (variable cible) ainsi que des ensembles de formation et de test.

Dans notre problème de vente de voitures, nous essayons d'utiliser les différentes caractéristiques de *car (X)* pour prédire son prix de *price (y).*

In [ ]:
```
# Create car sales X data (every column of car_sales except Price)
car_sales_X = ###

# Create car sales y data (the Price column of car_sales)
car_sales_y = ###
```

In [ ]:
```
# Use train_test_split to split the car_sales_X and car_sales_y data into
# training and test sets.
# Give the test set 20% of the data using the test_size parameter.
# For reproducibility set the random_state parameter to 42.
car_X_train, car_X_test, car_y_train, car_y_test = train_test_split(###,
                                                                    ###,
                                                                    test_size=###,
                                                                    random_state=###)

# Check the shapes of the training and test datasets
###
```

Combien de lignes y a-t-il dans chaque ensemble ?

Combien de colonnes y a-t-il dans chaque ensemble ?

D'accord, nos données sont divisées en ensembles d'entraînement et de test, il est temps de créer une petite boucle qui va :

1- Parcourez notre dictionnaire regression_models
2- Créer un Pipeline qui contient notre préprocesseur ainsi qu'un des modèles du dictionnaire
3- Adapte le pipeline aux données de formation à la vente de voitures
4- Évalue le modèle cible sur les données de test des ventes de voitures et ajoute les résultats à notre dictionnaire regression_results

In [ ]:
```
# Loop through the items in the regression_models dictionary
for model_name, model in regression_models.items():

    # Create a model Pipeline with a preprocessor step and model step
    model_pipeline = Pipeline(steps=[("preprocessor", ###),
                                     ("model", ###)])

    # Fit the model Pipeline to the car sales training data
    print(f"Fitting {model_name}...")
    model_pipeline.###(###, ###)

    # Score the model Pipeline on the test data appending the model_name to the
    # results dictionary
    print(f"Scoring {model_name}...")
    regression_results[model_name] = model_pipeline.score(###,
                                                           ###)
```

Nos modèles de régression ont été ajustés, voyons comment ils l'ont fait!

In [ ]:
```
# Check the results of each regression model by printing the regression_results
# dictionary
###
```

- Quel modèle a fait le mieux?
- Comment pourriez-vous améliorer ses résultats?
- Quelle métrique la méthode *score()* d'un modèle de régression renvoie-t-elle par défaut?

Puisque nous avons ajusté certains modèles mais que nous les avons comparés uniquement via la métrique par défaut contenue dans la méthode *score()* (score R ^ 2 ou coefficient de détermination), prenons le modèle RidgeRegression et évaluons-le avec quelques autres métriques de régression.

Plus précisément,nous trouvons:

1- **R^2 (pronounced r-squared) or coefficient of determination** - Compare les prédictions de vos modèles à la moyenne des cibles. Les valeurs peuvent aller de l'infini négatif (un modèle très médiocre) à 1. Par exemple, si votre modèle ne fait que prédire la moyenne des cibles, sa valeur R ^ 2 serait 0. Et si votre modèle prédit parfaitement une plage de nombres sa valeur R ^ 2 serait 1.
2- **Mean absolute error (MAE)** - La moyenne des différences absolues entre les prévisions et les valeurs réelles. Cela vous donne une idée de la fausseté de vos prédictions.
3- **Mean squared error (MSE)** - Différences moyennes au carré entre les prédictions et les valeurs réelles. La quadrature des erreurs supprime les erreurs négatives. Il amplifie également les valeurs aberrantes (échantillons qui ont des erreurs plus importantes).

Scikit-Learn a quelques classes intégrées qui vont nous aider avec celles-ci, à savoir, mean_absolute_error, mean_squared_error et r2_score.

In [ ]:
```
# Import mean_absolute_error from sklearn's metrics module
###

# Import mean_squared_error from sklearn's metrics module
###

# Import r2_score from sklearn's metrics module
###
```

Toutes les métriques d'évaluation qui nous intéressent comparent les prédictions d'un modèle avec les étiquettes. Sachant cela, nous devrons faire quelques prédictions.

Créons un pipeline avec le préprocesseur et un modèle *Ridge (),* adaptons-le aux données de formation sur les ventes de voitures, puis faisons des prédictions sur les données de test des ventes de voitures.

In [ ]:
```
# Create RidgeRegression Pipeline with preprocessor as the "preprocessor" and
# Ridge() as the "model".
ridge_pipeline = ###(steps=[("preprocessor", ###),
                                ("model", Ridge())])

# Fit the RidgeRegression Pipeline to the car sales training data
ridge_pipeline.fit(###, ###)

# Make predictions on the car sales test data using the RidgeRegression Pipeline
car_y_preds = ridge_pipeline.###(###)

# View the first 50 predictions
###
```

Bien! Maintenant, nous avons quelques prédictions, il est temps de les évaluer. Nous trouverons l'erreur quadratique moyenne (MSE), l'erreur absolue moyenne (MAE) et le score R ^ 2 (coefficient de détermination) de notre modèle.

In [ ]:
```
# EXAMPLE: Find the MSE by comparing the car sales test labels to the car sales predictions
mse = mean_squared_error(car_y_test, car_y_preds)
# Return the MSE
mse
```

In [ ]:
```
# Find the MAE by comparing the car sales test labels to the car sales predictions
###
# Return the MAE
###
```

In [ ]:
```
# Find the R^2 score by comparing the car sales test labels to the car sales predictions
###
# Return the R^2 score
###
```

Notre modèle pourrait potentiellement faire avec un réglage d'hyperparamètres (ce serait une excellente extension). Et nous pourrions probablement avoir besoin de trouver plus de données sur notre problème, 1000 lignes ne semblent pas être suffisantes.

Comment exporteriez-vous le modèle de régression entraîné?