Microsoft

SQL Server 2017

SQL Server 2017 自習書シリーズ No.3

SQL Server 2017 Machine Learning Services

Published: 2017 年 11 月 30 日
有限会社エスキューエル・クオリティ

SQLQuality
データベース コンサルティング

目次

STEP 1. SQL Server 2017 Machine Learning Services の概要

この STEP では、SQL Server の最新バージョンである「**SQL Server 2017**」で提供された「**Machine Learning Services**」（機械学習サービス）の概要を説明します。

この STEP では、次のことを学習します。

- ✓ SQL Server 2017 の主な新機能
- ✓ SQL Server 2017 Machine Learning Services の概要

1.1 SQL Server 2017 で提供された主な新機能

SQL Server の最新バージョンである「**SQL Server 2017**」は、2017 年 10 月に発売されました。1 つ前のバージョンである SQL Server 2016 の発売は 2016 年 6 月でしたので、わずか 1 年半弱でのバージョン アップになりますが、SQL Server 2017 には非常に多くの新機能が提供されています。その主なものは、次のとおりです。

	SQL Server 2017 からの主な新機能
Linux 対応	・**SQL Server 2017 on Linux** サポート プラットフォーム - Red Hat Enterprise Linux 7.3／7.4 Workstation, Server, and Desktop - SUSE Enterprise Linux Server v12 SP2 - Ubuntu 16.04 LTS - Docker Engine 1.8 以上 ・**on Linux で利用できる機能** - データベース エンジンに関するほぼすべての機能（詳細は 2章参照） - SQL Server Agent ジョブ（Transact-SQL の定期実行） - SSIS パッケージ（.dtsx）の実行
ビルトイン AI 機械学習	・**Python 統合**（Machine Learning Services） Python 言語をデータベース エンジンに統合（ビルトイン） Python における**ディープ ラーニング**（Deep Learning : 深層学習）の定番フレームワークである Microsoft Cognitive Toolkit（CNTK）や Chainer、TensorFlow、Caffe、Theano などを利用可能。GPU にも対応 ・**ネイティブ スコアリング**（PREDICT 関数による予測の実行）
注目の新機能	・**グラフ データベース**（Graph processing） ・**自動チューニング**（Automatic Tuning） ・**Adaptive Query Processing**（適応型クエリ処理） ・**クエリ ストア**の強化、**DTA** の強化、Scan/Read Ahead アルゴリズム改善 ・**列ストア インデックス**の強化（オンライン再構築や LOB 対応、性能向上etc） ・**インメモリ OLTP** の強化（制限緩和の追加や性能向上etc） ・**再開可能**なオンライン インデックス再構築（Resumable Rebuild Index） ・**AlwaysOn 可用性グループ**の強化（DTC 対応、クラスター レス構成など）
その他の新機能	・**Transact-SQL の強化** 新しい関数（TRIM、CONCAT_WS、TRANSLATE、STRING_AGG）、SELECT INTO でのファイル グループ指定、IDENTITY_CACHE オプションのサポート、日本語向けの新しい照合順序の追加など ・**セットアップ時の変更点**（tempdb ファイルの設定） ・**スマート バックアップ**、バックアップ性能の向上、Indirect Chekpoint ・**新しい DMV**（動的管理ビュー） ・**テンポラル テーブルの強化**（保持ポリシーの追加など） ・**XE プロファイラー**の提供、実行プランの検索機能 ・**SQL CLR** の**セキュリティ強化**
BI 関連の強化	・**Analysis Services の強化** Power Query Formula Language（M 言語）対応、ドリルスルー データでの DAX 指定、Ragged 階層対応、Object レベル セキュリティ、DAX 強化、DMV 強化、DISCOVER_CALC_DEPENDENCY など ・**Reporting Services の強化** レポート コメント、DAX エディター for SSDT、REST API 対応、軽量のインストーラー など ・**Integration Services の強化** SSIS パッケージのスケールアウト実行、Linux 対応 など ・**MDS（マスター データ サービス）の強化**

この中でも、目玉の新機能は、やはり**マルチ プラットフォーム**（**Linux** 対応）と、**ビルトイン AI**、**自動チューニング**（Automatic Tuning）、**グラフ データベース**対応などです。自動チューニングやグラフ データベースなどの新機能については、本自習書シリーズの「**No.1 SQL Server 2017 の新機能の概要**」編、Linux 対応については「**No.2 SQL Server 2017 on Linux**」編で詳しく説明しているので、こちらもぜひご覧いただければと思います。

1.2 Machine Learning Services（機械学習サービス）の概要

SQL Server 2017 から提供された「**Machine Learning Services**」（ML Services）は、**機械学習**（Machine Learning）を実現することができる機能です。

機械学習というと、スマート スピーカーやコルタナ、Siri といった**音声認識**アシスタント（AI アシスタント）や、**対話型ロボット**（AI ロボット）、自動車の**自動運転**、AlphaGo などのコンピューター囲碁、顔認証（顔パス）、自動翻訳といったコンシューマー向けのサービス、いわゆる **AI**（**人工知能**）を思い浮かべる方が多いと思います。もちろん、これらは、昨今のトレンドである**ディープ ラーニング**（Deep Learning : 深層学習）などを利用した画像認識や音声認識、自然言語処理を行うことによって実現することができます。

機械学習は、次のような利用用途もあります。

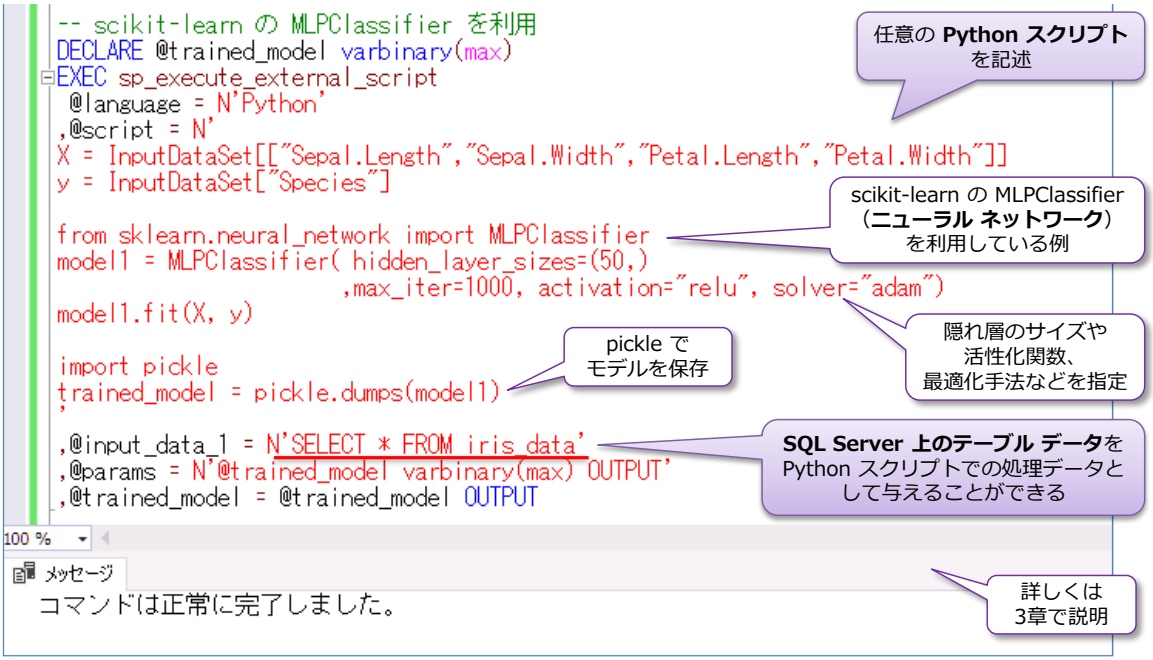
- 顧客のクラスタリング（セグメンテーション）
- クレジット カードの与信、不正使用の検知
- 商品の需要予測や、売上予測
- 商品のレコメンデーション（お勧め商品の提示）、クロスセリング、アップセリング
- DM（ダイレクト メール）や広告、キャンペーンなどの販促の効果測定
- 電力会社の電力消費量の予測
- 病気の検知（体のスキャン データなどを自動解析して、がんを早期発見するなど）
- 農業や、製造工場における規格外品の判別、品質管理（写真などを利用して、規格外の農作物／商品を除外、不良品を識別するなど）
- スマート ビルディングや農業での IoT センサーを利用した自動運用 など

データベース エンジニアにとっては、これらのほうが身近な例に感じるのではないでしょうか。こうした機械学習は、**特化型 AI**（特定の決まった作業を遂行するための AI）とも呼ばれているので、機械学習を利用したサービスであれば、「**AI を活用した ～ サービス**」と銘打つことができます。

SQL Server 2017 の Machine Learning Services（ビルトイン AI）

SQL Server 2017 からの新機能である **Machine Learning Services**（ML Services）は、SQL Server 2016 で提供されていた「**SQL Server R Services**」（R 統合）機能を強化して、名称変更したものです。SQL Server 2016 の「**SQL Server R Services**」では、R 言語を SQL Server に統合（ビルトイン）することによって、**機械学習**（Machine Learning）によるモデルの作成や予測（Predict）を行うことができましたが、SQL Server 2017 からは、**Python**（現在、機械学習およびデータ サイエンティストに最も人気があり、最も利用者数が多い言語）も統合されるようになりました。

次の画面は、SQL Server 2017 での Python 統合を利用している様子です。

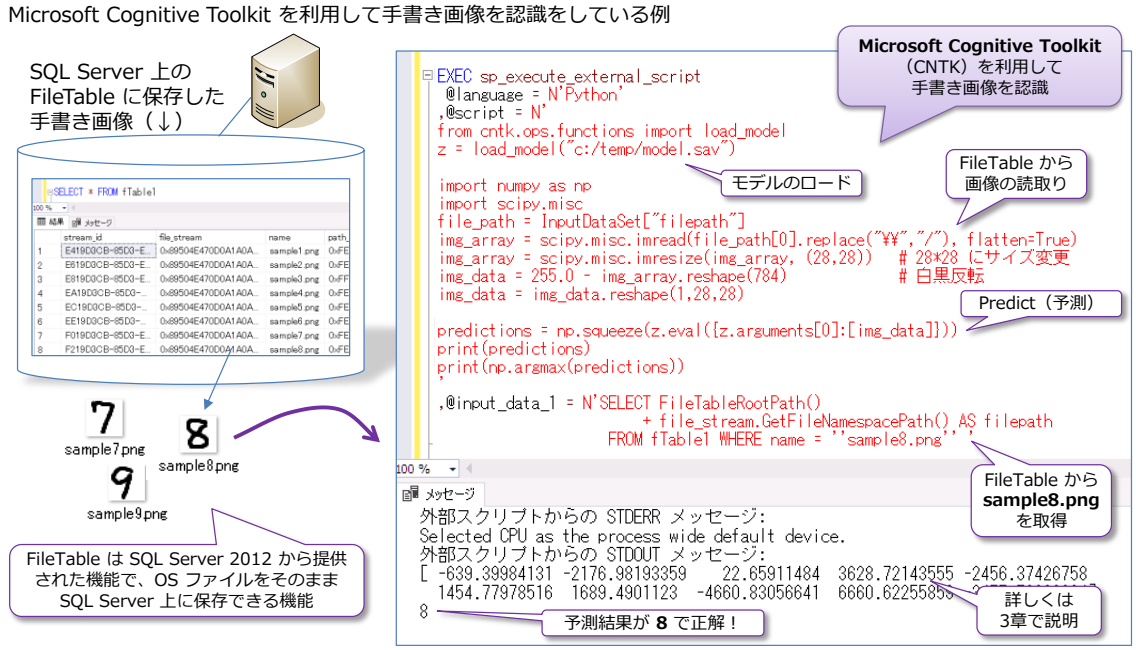


これは**ニューラル ネットワーク**を利用している例ですが、Python 統合では、**ディープ ラーニング**（深層学習）を利用した画像認識や音声認識、自然言語処理、各種の予測（Predict）およびモデル作成といった **AI**（人工知能）を実装することもできます（SQL Server は、初めて AI 機能を搭載／ビルトインしたデータベース製品でもあります）。

SQL Server 上に統合した **Python** では、Python の定番のライブラリである「**NumPy**」や「**pandas**」、「**scikit-learn**」、「**pickle**」、「**scipy**」、「**PIL**」などを利用できることはもちろんのこと、**ディープ ラーニング**での**定番フレームワーク**である「**Microsoft Cognitive Toolkit**（**CNTK**）」や「**Chainer**」、「**Google TensorFlow**」、「**Theano**」なども利用できます（通常の Python と同様、**pip** でインストールして利用できます）。

次の画面は、**Microsoft Cognitive Toolkit** を利用して画像を認識している場合の例です。

Microsoft Cognitive Toolkit を利用して手書き画像を認識をしている例



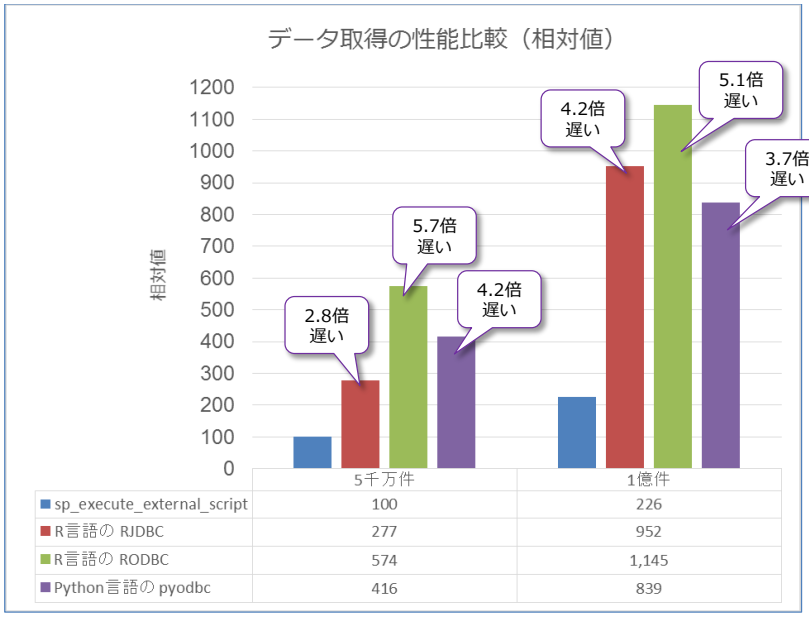
SQL Server 2017 上の FileTable に保存した手書きの画像データ（数字）を、Python 統合機能を利用して、予測しています。

ML Services の利点（SQL Server 上のデータを直接指定、高速取得）

SQL Server 2017 の **Machine Learning Services**（以降、**ML Services** と記述）の最大のメリットは、R または Python スクリプトで利用する**入力データ**（訓練データやテスト データ）に、**SELECT** ステートメントを記述して SQL Server 上のデータを直接指定できる点です。

通常の R や Python では、データベースのデータを利用するには、データベース サーバーへの接続やクエリを実行するためのスクリプトを記述しなければなりませんが、ML Services では、前出の画面のように、**sp_execute_external_script** というストアド プロシージャの「**@input_data_1**」引数に **SELECT** ステートメントを記述して、それをスクリプト内で利用することができます。

また、**sp_execute_external_script** ストアド プロシージャでのデータの取り込みは、通常の R や Python で利用されるデータ取得方法（R なら **RODBC** や **RJDBC**、Python なら **pyodbc**）よりも性能が良いというメリットがあります。次のグラフは、弊社環境で、データ件数が 5 千万件および 1 億件のデータに対して、データ取得を行ったときの性能を比較したものです。



■ テスト内容
5千万件および 1億件のデータに対して、単純なデータ取得を試みる。
全てのテストはローカルで実行

■ 使用したソフトウェア
・Windows Server 2016
・SQL Server 2017 Enterprise
・R 3.4.2
・Python 3.6
・Microsoft JDBC Driver 6.2
・Microsoft ODBC Driver 13.1

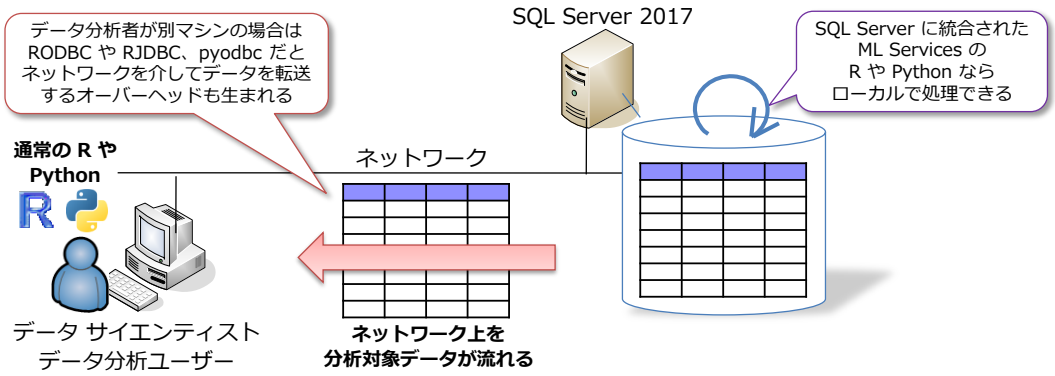
■ テスト環境
CPU：Core i7-6700K 4コア
メモリ： 64GB
ストレージ：Crucial MX200 1TB

＊ ベンチマークの結果の公表は、使用許諾契約書で禁止されているので、
sp_execute_external_script で 5千万件のデータを取得したときの値を 100 とした相対値で掲載しています。

ベンチマーク結果の公表は、使用許諾契約書で禁止されているので、5 千万件での **sp_execute_external_script** を利用した場合の結果を 100 とした相対値で表しています。5 千万件のデータ取得では、RJDBC だと 2.8 倍、RODBC は 5.7 倍、pyodbc は 4.2 倍も遅い結果であることを確認できました。

このテストは、ローカル環境（SQL Server と R および Python を同じマシン内）で実行したものですが、データ サイエンティスト／データ分析ユーザーがリモート環境であった場合は、さらにネットワーク転送の負荷も追加されることになります。

通常の R や Python ではデータ分析ユーザーがリモート マシンの場合はネットワーク転送の負荷もある

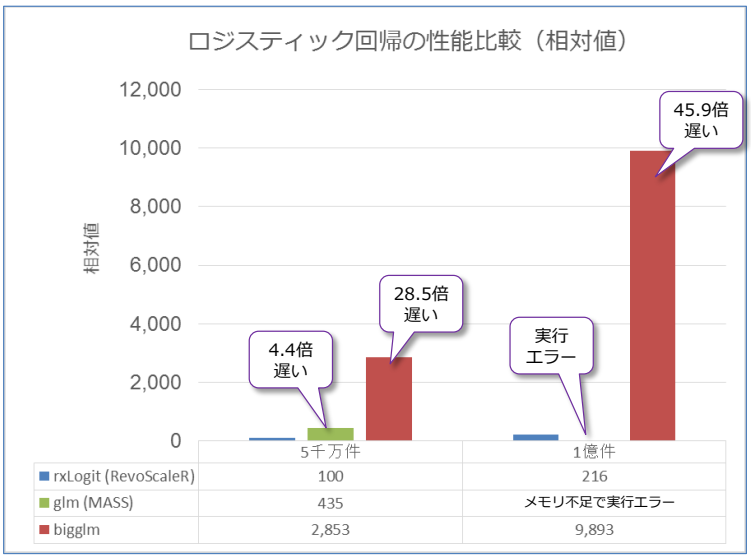


SQL Server 2017 の **ML Services** であれば、SQL Server 上のローカルでデータを処理することができるので、ネットワーク上をデータが流れるオーバーヘッドもなくなります。ストアド プロシージャを利用してモデルの作成や予測を実行することができるので、データ分析ユーザーは、実行結果のみを受け取れるようになります。

ML Services は、**GPU**（Graphics Processing Unit）にも対応しているので、SQL Server 上に搭載した GPU を利用すれば、CPU よりも高速に演算を行えるようになります。

エンタープライズ対応の R（RevoScaleR）

ML Services の前身は、SQL Server 2016 での「**SQL Server R Services**」になりますが、これは、オープンソースの R の性能的な欠点を補うために、Enterprise 向けに性能強化を計ったプラットフォームとして提供されていた「**Revolution R**」を買収したものでした（現在は、Microsoft R という名前に名称変更しています）。**Revolution R** での性能強化を計った R は、「**RevoScaleR**」と呼ばれていて、SQL Server 2017 の ML Services でも、この名前は残っています。次のグラフは、機械学習でよく利用されるアルゴリズムの 1 つである**ロジスティック回帰**（Logistic regression）を利用して、RevoScaleR の **rxLogit** と、通常の R で利用されることの多い **glm**（MASS ライブラリ）、**bigglm**（biglm ライブラリ）の性能を比較したものです。



■ テスト内容
5千万件および 1億件のデータに対して、ロジスティック回帰でのモデルを作成。説明変数は 4つ。

■ 使用したソフトウェア
・Windows Server 2016
・SQL Server 2017 Enterprise
・R 3.4.2
・Microsoft JDBC Driver 6.2

■ テスト環境
CPU：Core i7-3770K 4コア
メモリ： 32GB
ストレージ：Crucial M550 1TB

* ベンチマークの結果の公表は、使用許諾契約書で禁止されているので rxLogit で 5千万件の場合を 100 とした相対値で掲載

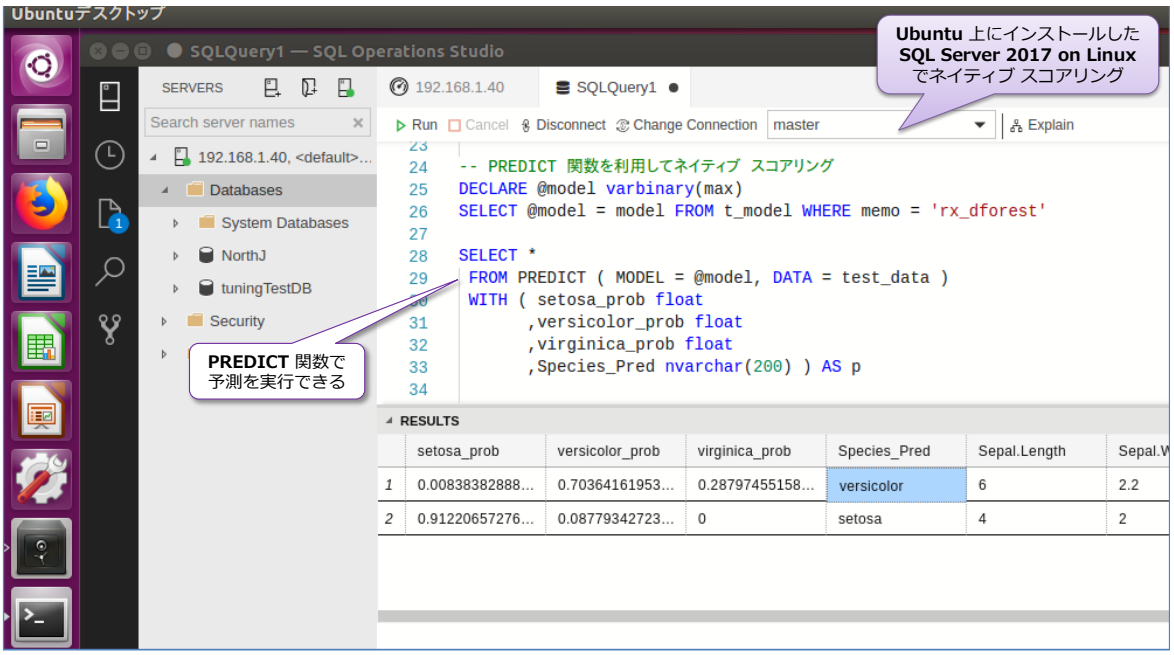
glm では、1 億件の場合にメモリ不足で実行エラーになっていますが、通常の R の関数はメモリに載っていることが前提で作られたものが多くあるので、大量のデータになった場合には、メモリ不足で実行できないケースが出てきます。

これに対して、**bigglm** および RevoScaleR の **rxLogit** は、大量データに対応するべく、メモリ使用量を抑えて実行できるように改良されたものですが、bigglm だと 5 千万件の場合に **28.5 倍**遅い結果、1 億件では **50 倍**も遅い結果になっています。このように、RevoScaleR では、大量データにも対応して、メモリ使用量も抑えられる関数が提供されているのも大きなメリットです。

ネイティブ スコアリング（PREDICT 関数）

RevoScaleR および RevoScaleR の Python 版である **Revoscalepy** で作成したモデル（**rxSerializeModel** という関数でシリアライズ化したモデル）は、Transact-SQL ステートメントの **PREDICT** 関数を利用してアクセスすることもできます。この機能は、Transact-SQL から（ストアド プロシージャを介さずに）ネイティブに利用できる（予測を実行できる）ことから「**ネイティブ スコアリング**」と呼ばれています。

ネイティブ スコアリング（**PREDICT** 関数）は、SQL Server 2017 on Linux でも利用することができます（以下の画面は、Ubuntu 上で SQL Server 2017 on Linux を動作させているマシンで PREDICT 関数を実行している例です）。



ネイティブ スコアリングの利用方法や、ML Services の基本的な利用方法については、次の章で詳しく説明します。

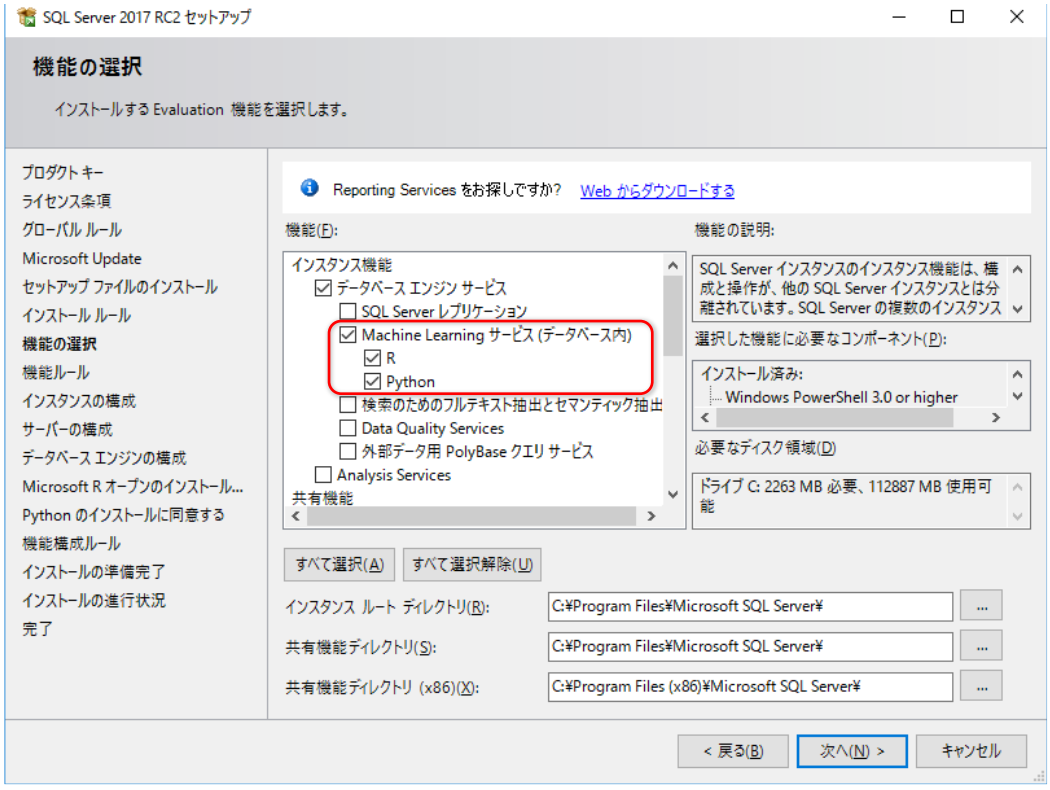
1.3 ML Services（Machine Learning Services）のインストール

SQL Server 2017 で、ML Services を利用するために必要となる作業は、次のとおりです。

- SQL Server のインストール時に「**Machine Learning サービス（データベース エンジン内）**」を選択して、「**R**」または「**Python**」を選択（両方選択しても OK）
- sp_configure で「**external scripts enabled**」を「**1**」に変更して、SQL Server サービスを再起動する

インストール手順は、次のとおりです。

1. まずは、SQL Server 2017 のインストール時に、次のように［**機能の選択**］ページで、［**データベース エンジン サービス**］の［**Machine Learning サービス（データベース エンジン内）**］をチェックして、「**R**」や「**Python**」（利用したい言語）をチェックします。

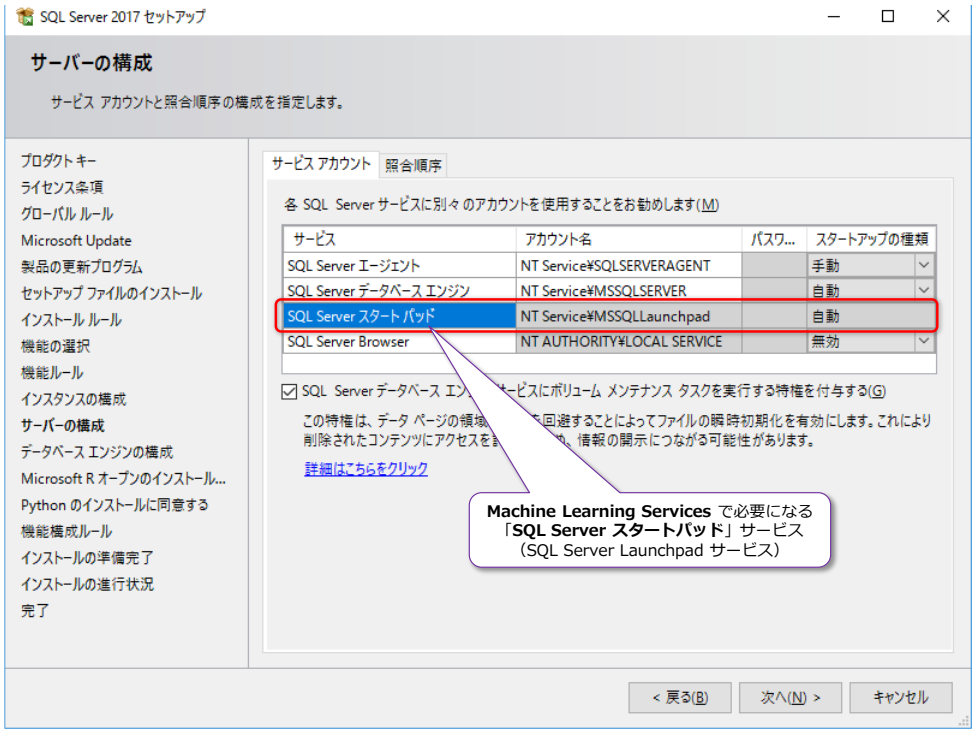


この自習書では、R も Python もどちらも利用するので、どちらもチェックしておいてください。

なお、Machine Learning Services は、ドメイン コントローラーにインストールすることはできません。また、スタンドアロン版の Machine Learning Server との共存はお勧めではありません。そうしたインストールにあたっての詳細については、SQL Server のヘルプの以下のトピックが参考になります。

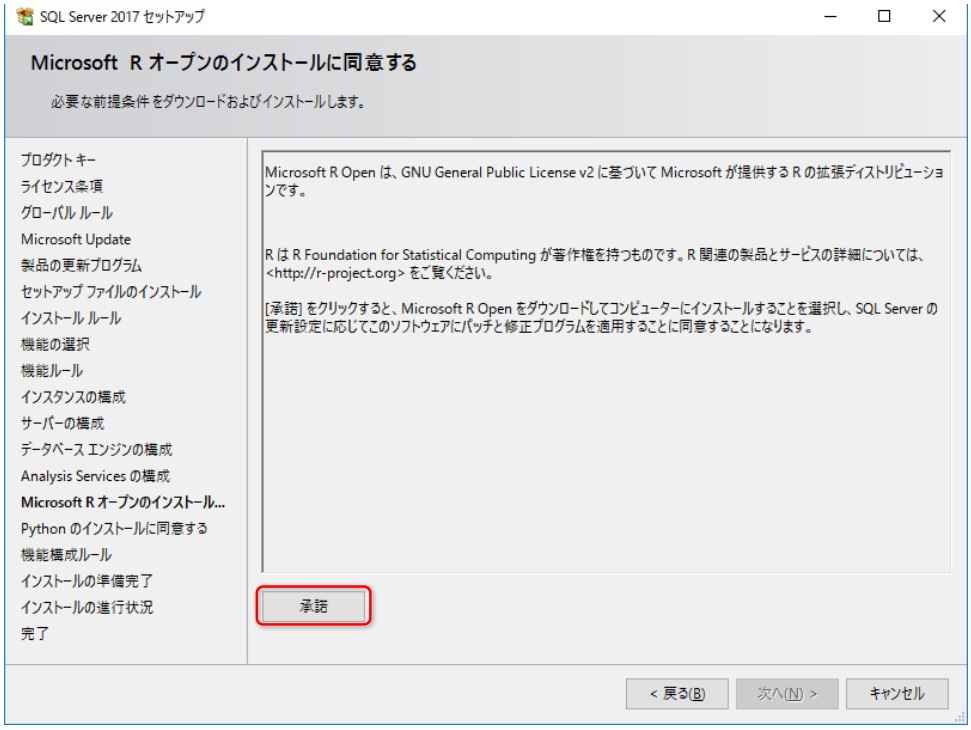
SQL Server マシン ラーニング Services (In-database) セットアップします
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/r/set-up-sql-server-r-services-in-database

［**機能の選択**］ページで R や Python をチェックしている場合は、次のように［**サーバーの構成**］ページで「**SQL Server スタートパッド**」サービス（**SQL Server Launchpad** サービス）が表示されるようになります。



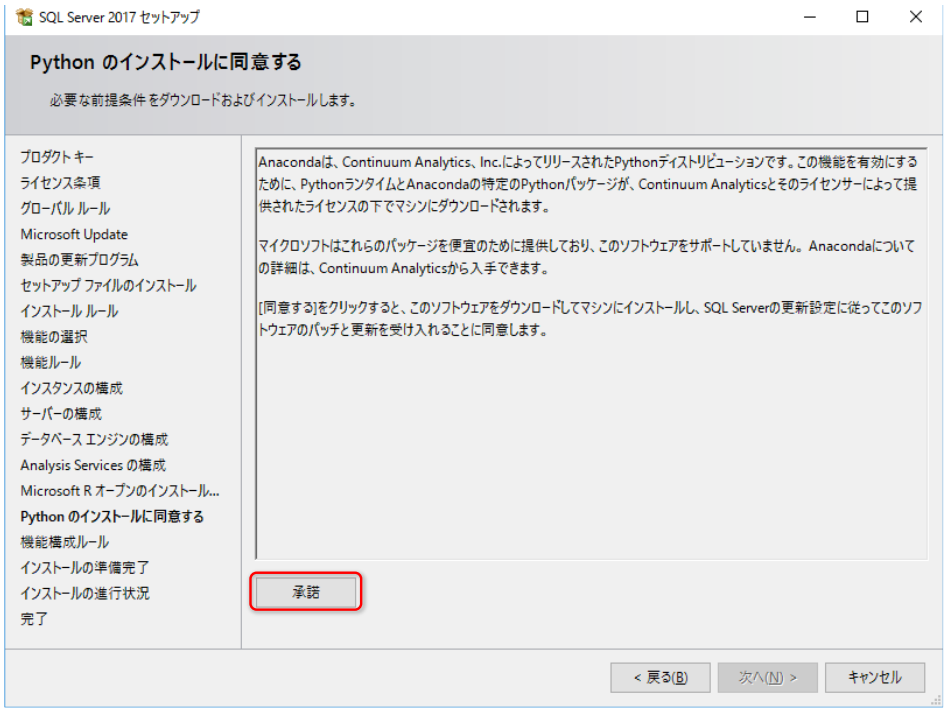
このサービスは、ML Services を利用するために必要になるので、［**スタートアップの種類**］が「**自動**」になっていることを確認します（自動起動するように設定します）。

ML Services の利用言語として「**R**」を選択している場合は、次のように［**Microsoft R オープンのインストールに同意する**］ページも表示されます。



内容を確認した上で、[**承諾**] ボタンをクリックすれば、SQL Server に統合された R 言語を利用できるようになります。

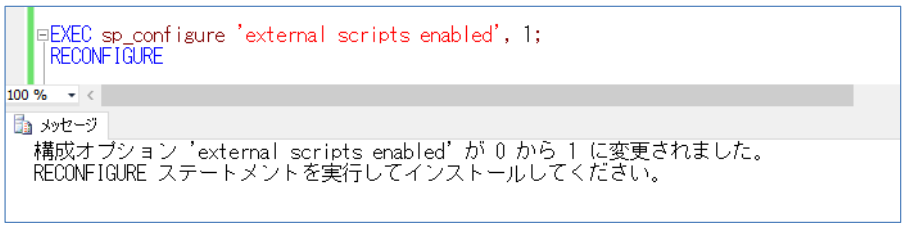
ML Services の利用言語として「**Python**」を選択している場合は、次のように［**Python のインストールに同意する**］ページも表示されます。



内容を確認した上で、［**承諾**］ボタンをクリックすれば、SQL Server に統合された Python を利用できるようになります（Python 統合では **Anaconda** ディストリビューションを利用しています）。

2. ML Services のインストールが完了した後は、**sp_configure** を利用して、「**external scripts enabled**」を「**1**」に変更します。これを行うには、Management Studio を起動して、クエリ エディターを開き、次のように実行します。

```
EXEC sp_configure 'external scripts enabled', 1;
RECONFIGURE
```



3. 次に、SQL Server サービスを再起動します。

以上で、ML Services を利用できるようになり、SQL Server に統合（ビルトイン）された Python や R をクエリ エディターから実行できるようになります（スクリプトの実行には、後述の **sp_execute_external_script** システム ストアド プロシージャを利用します）。

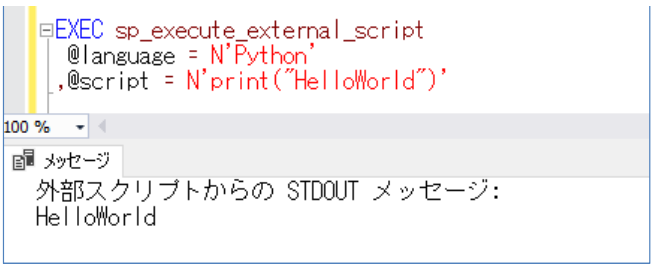
Python スクリプトの実行 ～sp_execute_external_script～

クエリ エディターから Python スクリプトを実行するには、次のように **sp_execute_external_script** システム ストアド プロシージャを利用します。

```
EXEC sp_execute_external_script
       @language = N'Python'
     , @script = N'実行したい Python スクリプト'
     , @input_data_1 = N'Python で処理したい入力データ'
     , @input_data_1_name = N'input_data_1 に対して設定する名前。既定値は InputDataSet'
     , @output_data_1_name = N'Python スクリプトで処理した結果。既定値は OutputDataSet'
     , @params = N'追加の変数定義。結果を変数で受け取る場合などに利用'
     WITH RESULT SETS ((列名 データ型 null/not null, ･･･));
```

@language で「**Python**」を指定して、**@script** に「**Python スクリプト**」を記述することで、任意の Python スクリプトを実行することができます。また、Python スクリプトに与えたい入力データは「**@input_data_1**」および「**@input_data_1_name**」で指定し、Python スクリプトで処理した結果（出力データ）は「**@output_data_1_name**」で指定できます。「**@input_data_1_name**」を省略した場合は **InputDataSet**、「**@output_data_1_name**」を省略した場合は **OutputDataSet** という名前が補われます。

また、最小限のパラメーターは、**@language** と **@script** の 2 つで、以下のように実行することもできます（Python の print で単純に文字列を出力）。



R スクリプトの実行

sp_execute_external_script では、**@language** で「**R**」を指定することで、R スクリプトを実行することができます（その他の引数の利用方法は Python の場合と同様です）。

```
EXEC sp_execute_external_script
       @language = N'R'
     , @script = N'実行したい R スクリプト'
     , @input_data_1 = N'R で処理したい入力データ'
     , ～
```

次の章では、このストアド プロシージャの利用方法を具体的に説明します。

STEP 2. 機械学習の基礎

この STEP では、SQL Server 2017 Machine Learning Services を利用して、機械学習を行う上での基本的な操作方法を説明します。「**決定木**」や「**ランダム フォレスト**」など、簡単に利用できる機械学習のアルゴリズムを利用して、モデルの作成方法や予測（Predict）の仕方、モデルの保存方法、ネイティブ スコアリングなどを説明します。

この STEP では、次のことを学習します。

- ✓ 利用するデータセット（R 言語に組み込まれた iris データ）
- ✓ 決定木（Decision Tree）のモデル作成と予測（Predict）
- ✓ ランダム フォレスト（Random Forest）のモデル作成と予測
- ✓ R での外部パッケージの利用（CREATE EXTERNAL LIBRARY）
- ✓ RevoScaleR の rxDTree、rxDForest
- ✓ SQL Server のデータを利用してモデル作成（@input_data_1）
- ✓ Python（Revoscalepy）でのランダム フォレスト（rx_dforest）
- ✓ モデルの保存（rxSerializeModel、rx_serialize_model）
- ✓ ネイティブ スコアリング（PREDICT 関数）
- ✓ トレーニング データとテスト データの分割

2.1 機械学習の基本

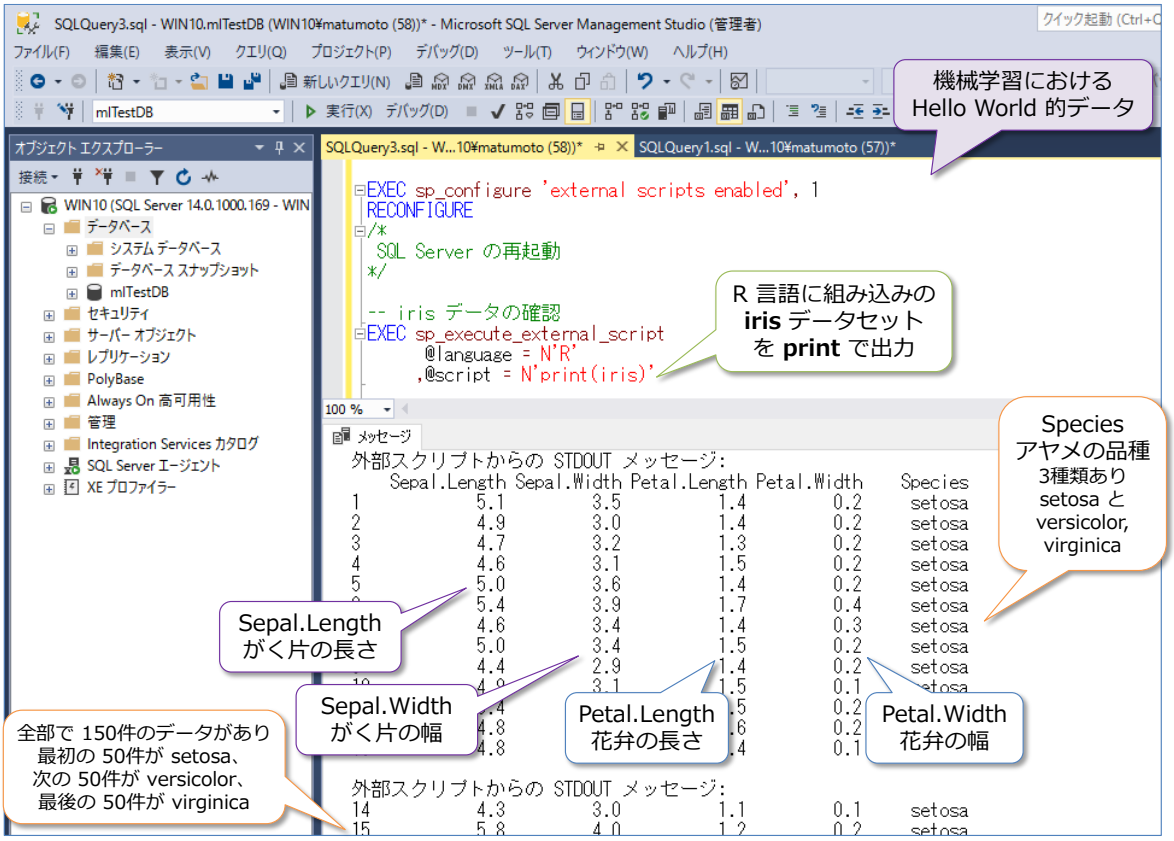
機械学習では、決定木（Decision Tree）やランダム フォレスト、ロジスティック回帰、ナイーブベイズ、サポート ベクター マシン（SVM）、確率的勾配降下法（SGD）、ニューラル ネットワーク、ディープ ラーニングなど、さまざまなアルゴリズムがありますが、R および Python では、これらのアルゴリズムを簡単に利用できるライブラリ／パッケージが用意されています。

また、1 章で紹介した **RevoScaleR**（R の性能強化を計ったライブラリ）では、**rxDTree**（決定木）や、**rxDForest**（ランダム フォレスト）、**rxLogit**（ロジスティック回帰）、**rxNaiveBayes**（ナイーブベイズ）などの関数が提供されています。どの関数を利用する場合でも、基本的な利用方法は同じになるので、この章では、決定木とランダム フォレストを利用して、データを予測する方法を説明しながら、機械学習の基本となる部分を説明します。

この章で利用するデータ（iris：アヤメ）

この章では、R を利用している方にはお馴染みの「**iris**」データを利用します。**iris** は、R 言語に含まれた（組み込まれている）、学習用のサンプル データセットで、**アヤメ**の **Sepal**（がく片）や **Petal**（花弁）の長さ／幅が格納されているもので、次のようにデータを確認できます。

```
-- iris データの確認（print で出力）
EXEC sp_execute_external_script
      @language = N'R'
     ,@script = N'print(iris)'
```

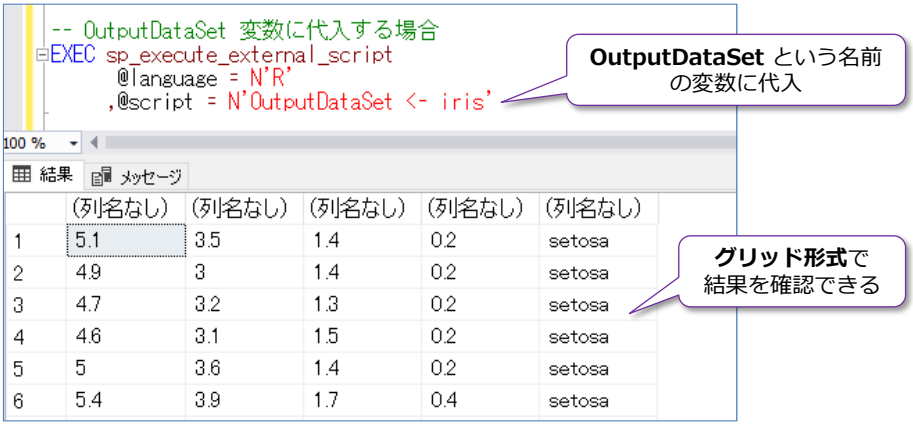


このデータは、植物生物学者である Edgar Anderson 氏がアヤメ（iris）から収集した「**がく片**」（**Sepal**）と「**花弁**」（**Petal**）の**長さ**（**Length**）と**幅**（**Width**）になっています（品種ごとに 50 の花から収集したデータ）。**Species** 列には、アヤメの品種が格納されています。

sp_execute_external_script ストアド プロシージャでは、**@language** に「**R**」を指定して、**@script** に「**print(iris)**」と指定することで、**iris** データセットの中身を **print** で標準出力（Management Studio では「**メッセージ**」タブ）に出力しています。

sp_execute_external_script ストアド プロシージャは、次のように「**OutputDataSet**」という名前の変数に値を代入することで、グリッド形式で結果を確認することもできます（R 言語では、大文字と小文字を区別するので、**O**utput**D**ata**S**et の **O** と **D** と **S** は大文字、残りは小文字で入力することに注意してください）。

```
-- OutputDataSet 変数に代入する場合
EXEC sp_execute_external_script
      @language = N'R'
     ,@script = N'OutputDataSet <- iris'
```

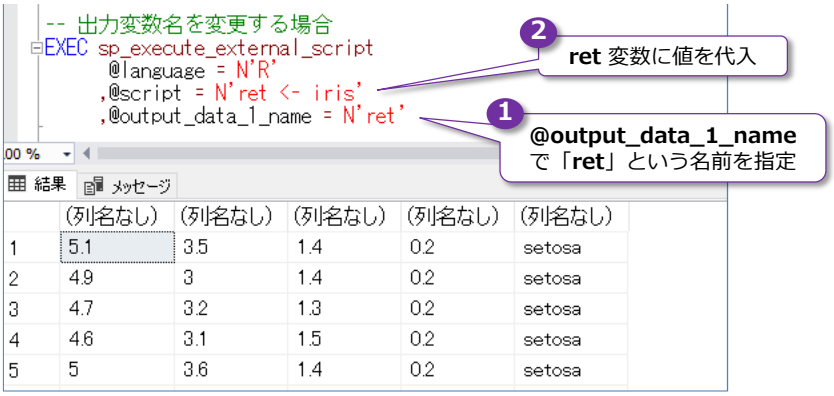


	(列名なし)	(列名なし)	(列名なし)	(列名なし)	(列名なし)
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

R 言語では、値の代入は「**=**」ではなく「**<-**」を利用するので、「**OutputDataSet <- iris**」と記述して、iris データを **OutputDataSet** 変数に代入しています。

なお、次のように「**@output_data_1_name**」パラメーターに出力変数名を指定（以下は **ret** という名前を指定）すれば、別の名前の変数に値を代入して、結果を表示することもできます。

```
-- 出力変数名を「ret」に変更する場合
EXEC sp_execute_external_script
      @language = N'R'
     ,@script = N'ret <- iris'
     ,@output_data_1_name = N'ret'
```



	(列名なし)	(列名なし)	(列名なし)	(列名なし)	(列名なし)
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa

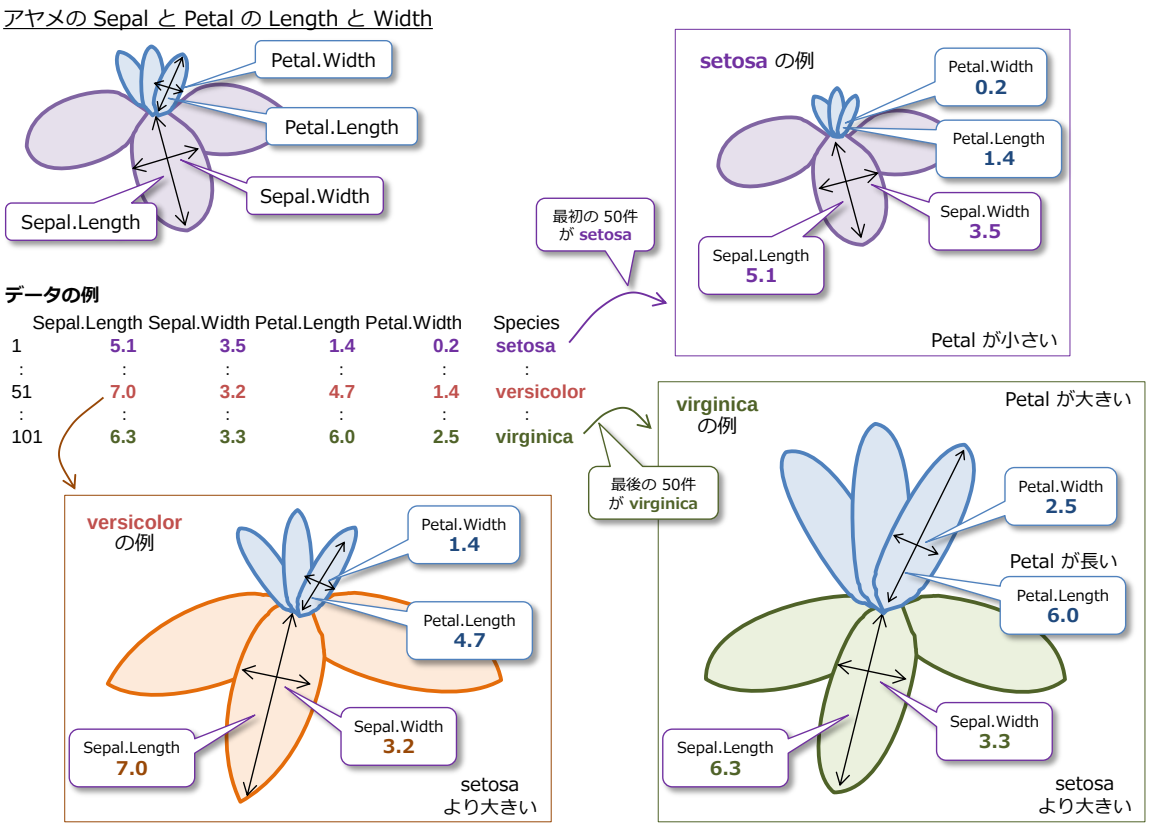
OutputDataSet や、ここで指定した **ret** は、「**出力変数**」と呼ばれますが、出力変数に代入できる値は、**データ フレーム**形式（R 言語の場合は data.frame、Python 言語の場合は pandas.DataFrame 形式）のデータのみになります。R 言語に組み込みの iris データは、データフレーム形式なので、上記のような出力が可能です。R や Python スクリプトで処理した結果で、データ フレーム形式にならないもの（後述のモデル作成でのバイナリ データなど）を出力したい場合には、「**@params**」パラメーターを利用して、変数を定義することで出力できるようになりますが、これについては後述します。

アヤメ（iris）の Sepal と Petal

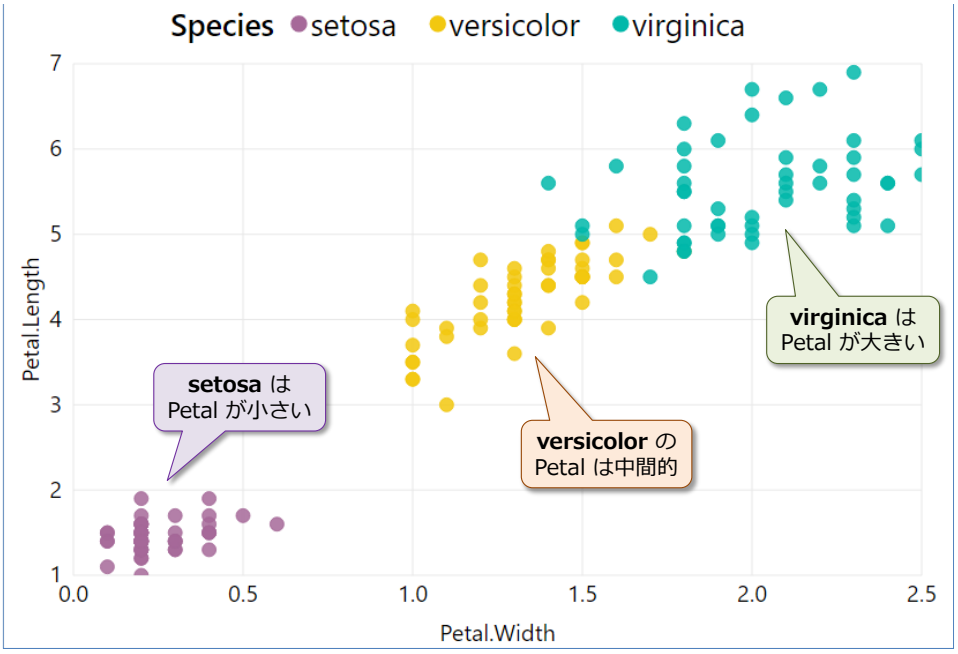
この章では、このアヤメ（iris）のデータを利用して、モデルを作成し、そのモデルを使って種類の予測（Predict）を行ってみます。アヤメの **Sepal**（がく片）と **Petal**（花弁）は、次の場所です。

アヤメの Sepal と Petal

R 言語の **iris** データセットには、**Sepal** と **Petal** の**長さ**（**Length**）と**幅**（**Width**）、**種類**（**Species**）が格納されていて、それぞれのデータのイメージは次のようになります。



データの件数は全部で **150 件**で、最初の 50 件が **setosa**、次の 50 件が **versicolor**、最後の 50 件が **virginica** というアヤメの種類になっています。**setosa** は、Petal が小さく、**virginica** は Petal が大きいという特徴があり、以下のグラフは、この 150 件のデータを **Petal.Length** を縦軸、**Petal.Width** を横軸にして、散布図で表現したものです。



以降では、このデータを利用して、機械学習（Machine Learning）を行っていきます。150 件のデータを訓練データ（トレーニング データ）として、機械学習（決定木やランダム フォレストなど）を利用してモデルを作成し、種類を予測（Predict）してみます。

2.2 決定木（Decision Tree：ディシジョン ツリー）

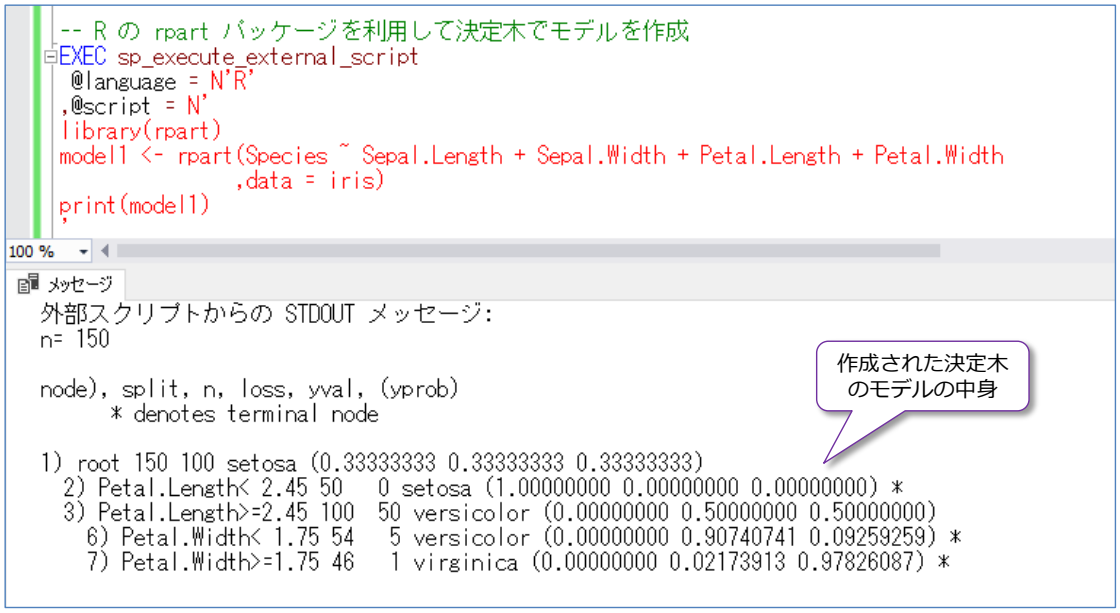
まずは、一番分かりやすい機械学習の例として、**決定木**（Decision Tree：ディシジョン ツリー）を利用してみます。決定木そのものを実際の業務データで予測に用いることはほとんどないのですが、後述の実務でよく利用する**ランダム フォレスト**のベースになっていたり、他の機械学習のアルゴリズムとの比較をするための基本アルゴリズムとして覚えておくと便利です。

Let's Try

それでは、**決定木**を試してみましょう。ここでは **R 言語**を利用した方法を説明しますが、わずか数行のコードで試せるものばかりなので、R が初めての方でも、Python しか利用したことのない方でもぜひ試してみてください（Python での利用方法については、この章の後半で説明します）。

1. R では、**rpart**（Recursive Partitioning and Regression Trees）というパッケージを利用することで、簡単に決定木のモデルを作成することができます。まずは、次のようにスクリプトを記述して、実行してみてください（大文字と小文字に注意して入力してください）。

```
-- R の rpart パッケージを利用して決定木のモデルを作成（アヤメの種類予測のモデル）
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
library(rpart)
model1 <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                ,data = iris)
print(model1)
'
```



R では、パッケージを利用する場合に「**library(パッケージ名)**」と記述するので、「**library(rpart)**」として、**rpart** パッケージを利用しています。このパッケージでは、**rpart**

関数で決定木のモデルを作成できますが、「**rpart(formula, data=データ)**」という形で利用します。今回、データには **iris** データセットを指定しています。**formura** は、**式**という意味ですが、R 言語での機械学習では、次のように利用します。

```
目的変数 ~ 説明変数１ + 説明変数２ + 説明変数３ + …
```

目的変数は、実際に予測（Predict）したい対象で、**説明変数**は、予測のために利用する情報になり、今回は、次のように指定していました。

```
model1 <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                ,data = iris)
```

アヤメの種類を Sepal と Petal の長さや幅で予測するためのモデル

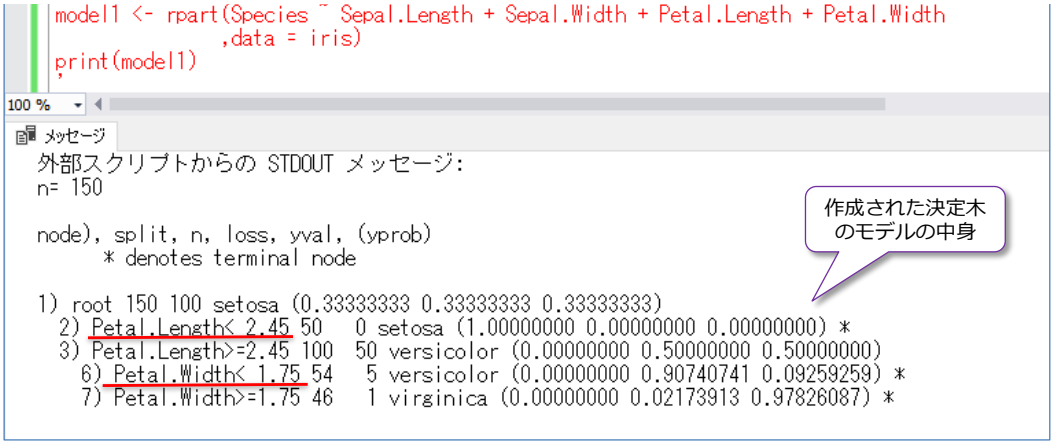
Species（アヤメの種類）を目的変数にして、**Sepal.Length** と **Sepal.Width**、**Petal.Length**、**Petal.Width** の 4 つを説明変数に指定して、アヤメの種類を予測するためのモデルを作成しています。

Species（目的変数）の後の「**~**」は**チルダ**で、チルダは、キーボードの右上の「**へ**」のキーを Shift キーを押しながら押下することで入力できます。説明変数は、**+** 演算子で、複数の変数を指定できます（予測に利用したい変数の数だけ + で追加していきます）。

rpart の結果（決定木のモデル）は、「**model1 <-**」とすることで **model1** という名前の変数に代入しています。

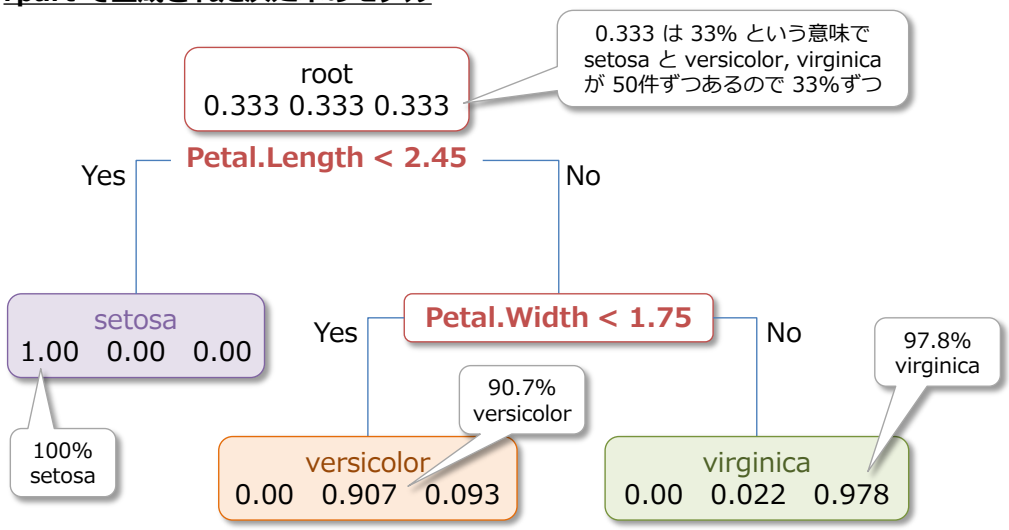
モデル（決定木）の中身

実際に作成されたモデルは、「**print(model1)**」と記述することで、「**メッセージ**」タブに出力しています。



結果のうち、「**Petal.Length<2.45**」や「**Petal.Width<1.75**」という記述があることを確認できます。これを図で表現すると、次のようになります。

rpart で生成された決定木のモデル



出力されたモデルの中には、3 つの数値「**0.3333… 0.3333… 0.333…**」があり、**root** は、アヤメの種類が 3 種類（左から setosa, versicolor, virginica の順）で、150 件のうち、50 件ずつのデータが入っているので、33.3% が 3 つ並んでいます。

次に、「**Petal.Length<2.45**」という分岐があり、Petal の Length が **2.45** より小さいなら（Yes なら）、「**100% setosa**」になります。今回の iris データセットにある setosa のデータは、すべて Petal の Length が **2.45** より小さくて、また versicolor と virginica には、Petal の Length が 2.45 より小さいものが存在していないがために、このようなモデルが生成されています。

「**Petal.Length<2.45**」が **No** の場合、つまり「**Petal.Length>=2.45**」の場合は、versicolor か virginica になりますが、もう 1 つ分岐があり「**Petal.Width<1.75**」となっています。この分岐は、Petal の Width が **1.75** より小さいなら（Yes なら）「**90.7%** が **versicolor**」、**1.75** 以上なら「**97.8%** が **virginica**」になります。

このように、決定木では、説明変数で指定された値（**Sepal.Length** と **Sepal.Width**、**Petal.Length**、**Petal.Width** の 4 つの値）から、データを分類できないかを検討して、ツリー（値による分岐）を作成します。今回利用した 150 件の iris データでは、**Sepal** に関しては、ツリーに含まれませんでしたが、後述のランダム フォレストなどでは、Sepal の値も考慮したツリーが作成されたりします。

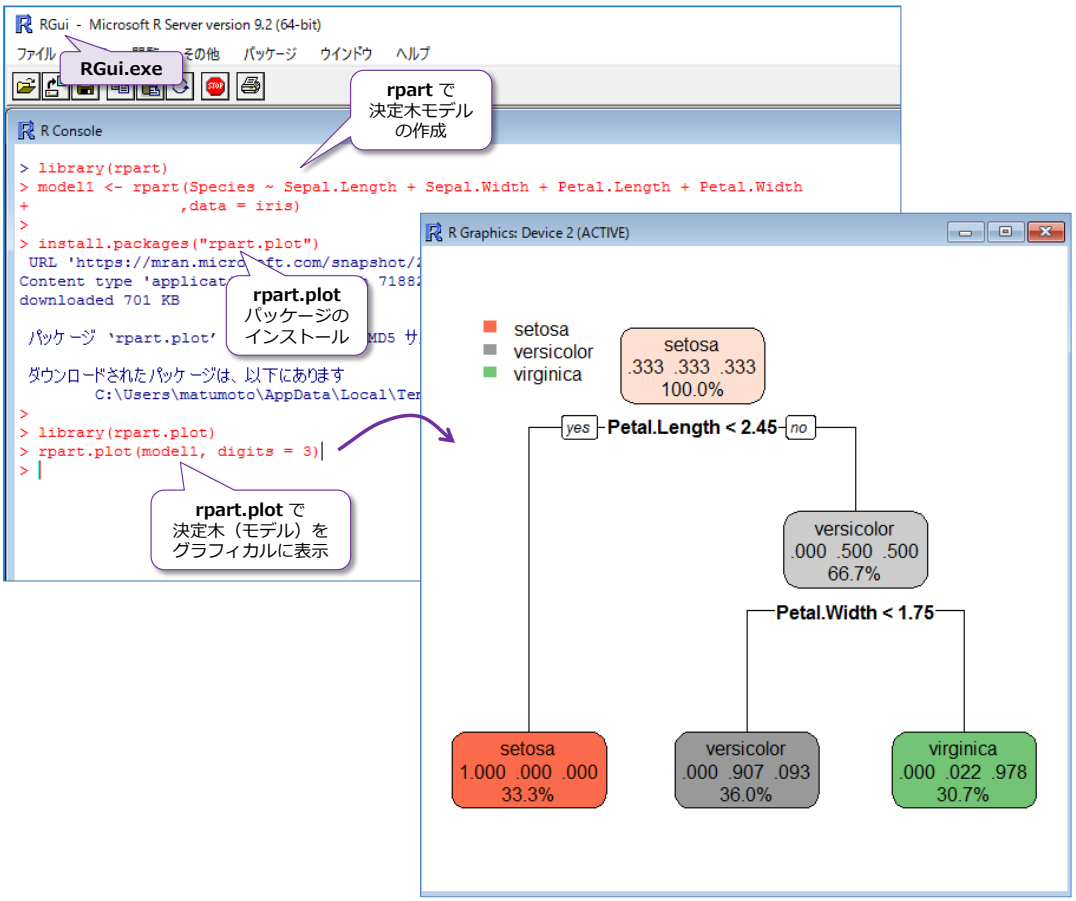
Note： 決定木モデルの可視化（R の rpart.plot パッケージ）
rpart で作成した決定木のモデルは、**rpart.plot** パッケージを利用することで、可視化（ツリーをグラフィカルに表示）することもできます。

これを行うには、R スクリプトを実行できる「**RGui.exe**」や「**RStudio**」ツールを利用しますが、SQL Server 2017 の Machine Learning Services をインストールすると、「**RGui.exe**」に関しては、以下のフォルダーにインストールされています（既定のインスタンスの場合）。

C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥R_SERVICES¥bin¥x64

RGui.exe は、R スクリプトを実行できる GUI ツールで、次のように R スクリプトを記述することで、rpart で作成した決定木のモデルをグラフィカルに表示することができます。

```
# rpart で決定木モデルの作成
library(rpart)
model1 <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                ,data = iris)

# rpart.plot パッケージのインストール
install.packages("rpart.plot")
library(rpart.plot)

# rpart.plot でモデルをグラフィカルに表示
rpart.plot(model1, digits = 3)
```



予測（Predict）

次に、作成した決定木のモデルを利用して、種類の**予測**（**Predict**）を行ってみましょう。予測を行うには、**predict** 関数を利用しますが、予測では、モデルを作成したときと同じ形式のデータ（列名や列数、データ型など）を与える必要があります（列名は、大文字と小文字も区別します）。

1. ここでは、次のように入力して、予測を実行してみます。**#** の部分は、R 言語での**コメント**になるので、省略してかまいません。また、モデルを出力するところ（**print(model1)**）までは、前の手順と同じスクリプトで、「**df1 <- ～**」以降が追加するスクリプトになります。

```
-- predict 関数で予測
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
library(rpart)
model1 <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
               ,data = iris)
print(model1)

# 予測させたいデータを data.frame（データ フレーム形式）で指定
# Petal.Length が 2.45 より小さいもの
df1 <- data.frame(Sepal.Length = 6.0
                 ,Sepal.Width  = 2.2
                 ,Petal.Length = 2.0
                 ,Petal.Width  = 1.5
                 ,Species = "dummy")

# predict 関数で予測
print(predict(model1, df1))
'
```

df1 は変数名で、モデル作成時に利用した **iris** データは、**データ フレーム形式**（リレーショナル データベースにおけるテーブルに相当するデータ形式）なので、**data.frame** を利用して、データ フレームを作成しています。列名には、**iris** データと同様、**Sepal.Length** と **Sepal.Width**、**Petal.Length**、**Petal.Width**、**Species** の 5 つの列を指定します（大文字と小文字を区別するので、すべて正しく指定する必要があります）。

「**列名 = 値**」で、データ フレームに格納する値を指定しますが、「**6.0**」や「**2.2**」、「**2.0**」、「**1.5**」などを指定しています。ここでは、意図的に **Petal.Length** の値が「**2.45**」より小さい値になるように「**2.0**」を指定しています。種類（**Species**）に関しては利用しないので（種類は **predict** 関数で予測をするので）、**dummy** にしています。

predict 関数は、「**predict(モデル名, データ)**」のように利用するので、モデル名に「**model1**」、データに「**df1**」（上で作成したデータ フレーム）を指定しています。

predict（予測）の結果は、**print** 関数で出力して、次のような結果になります。

```
-- predict 関数で予測
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
library(rpart)
model1 <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                ,data = iris)
print(model1)

# 予測させたいデータを data.frame (データ フレーム形式) で指定
# Petal.Length が 2.45 より小さいもの
df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 2.0
                  ,Petal.Width  = 1.5
                  ,Species = "dummy")

# predict 関数で予測
print(predict(model1, df1))
'
```

Petal.Length が 2.45 より小さいデータ

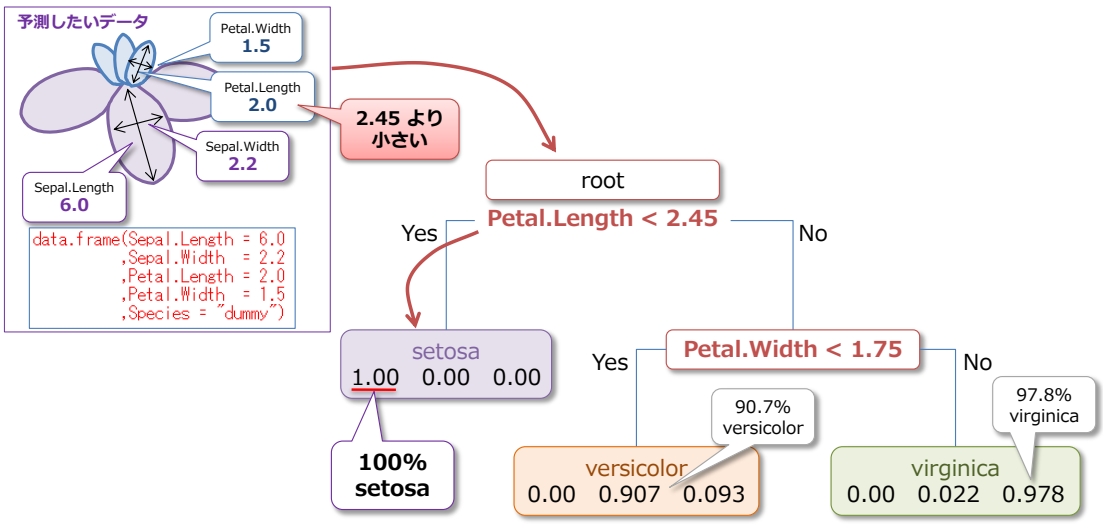
```
00 %
メッセージ

1) root 150 100 setosa (0.33333333 0.33333333 0.33333333)
  2) Petal.Length< 2.45 50   0 setosa (1.00000000 0.00000000 0.00000000) *
  3) Petal.Length>=2.45 100  50 versicolor (0.00000000 0.50000000 0.50000000)
    6) Petal.Width< 1.75 54   5 ver                000 0.90740741 0.09259259) *
    7) Petal.Width>=1.75 46   1 vir                00 0.02173913 0.97826087) *
  setosa versicolor virginica
1      1          0         0
```

モデル（決定木）の中身

predict（予測）の結果

Petal.Length に「**2.0**」を指定して、「**2.45**」よりも小さいデータを与えているので、結果は **setosa** が **1** と表示されて、「**100% setosa**」であると予測されています。これは、次のように解釈されています。

rpart で生成された決定木のモデルで予測（predict）



このように、**Petal.Length** が「**2.45**」より小さいだけで、**setosa** と判定されるので（モデルの作成に利用した 150 件のデータがそういうデータであったため）、極端なことを言うと、次のようなデータを与えても、**setosa** と判定されます。

```
# 極端なデータを与えて予測を実行すると...
df1 <- data.frame(Sepal.Length = 99.0
                  ,Sepal.Width  = 99.0
                  ,Petal.Length = 2.0
```

```
                    ,Petal.Width  = 99.0
                    ,Species = "dummy")
```

```
df1 <- data.frame(Sepal.Length = 99.0
                  ,Sepal.Width  = 99.0
                  ,Petal.Length = 2.0
                  ,Petal.Width  = 99.0
                  ,Species = "dummy")
print(predict(model1, df1))
```

```
100 %
メッセージ
外部スクリプトからの STDOUT メッセージ:
  setosa versicolor virginica
1      1          0         0
```

Petal.Length が **2.45** より小さいので **100% setosa** と判定される

2. 次に、予測させたいデータの **Petal.Length** を「**5.0**」に変更して、実行してみます。

```
df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.5
                  ,Species = "dummy")
```

```
-- Petal.Length = 5.0 に変更
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
library(rpart)
model1 <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                ,data = iris)
# print(model1)

df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.5
                  ,Species = "dummy")

print(predict(model1, df1))
'
100 %
メッセージ
外部スクリプトからの STDOUT メッセージ:
  setosa versicolor  virginica
1      0  0.9074074 0.09259259
```

Petal.Length を **5.0** に変更

Petal.Width は **1.5** を指定

Petal.Length が **2.45** 以上で **Petal.Width** が **1.75** より小さいので **versicolor** と判定される

結果は、**90.7% versicolor** と判定されます。今回作成された決定木では、**Petal.Length** が **2.45** 以上で、**Petal.Width** が **1.75** より小さい場合は、versicolor と判定されるので、このような結果になっています。

実は、ここで指定した「**6.0、2.2、5.0、1.5**」という値は、実際の **iris** データの中にある、120 件目のものと全く同じ数値です。

```
-- iris データ
EXEC sp_execute_external_script
      @language = N'R'
     ,@script = N'OutputDataSet <- iris'
```

100 %

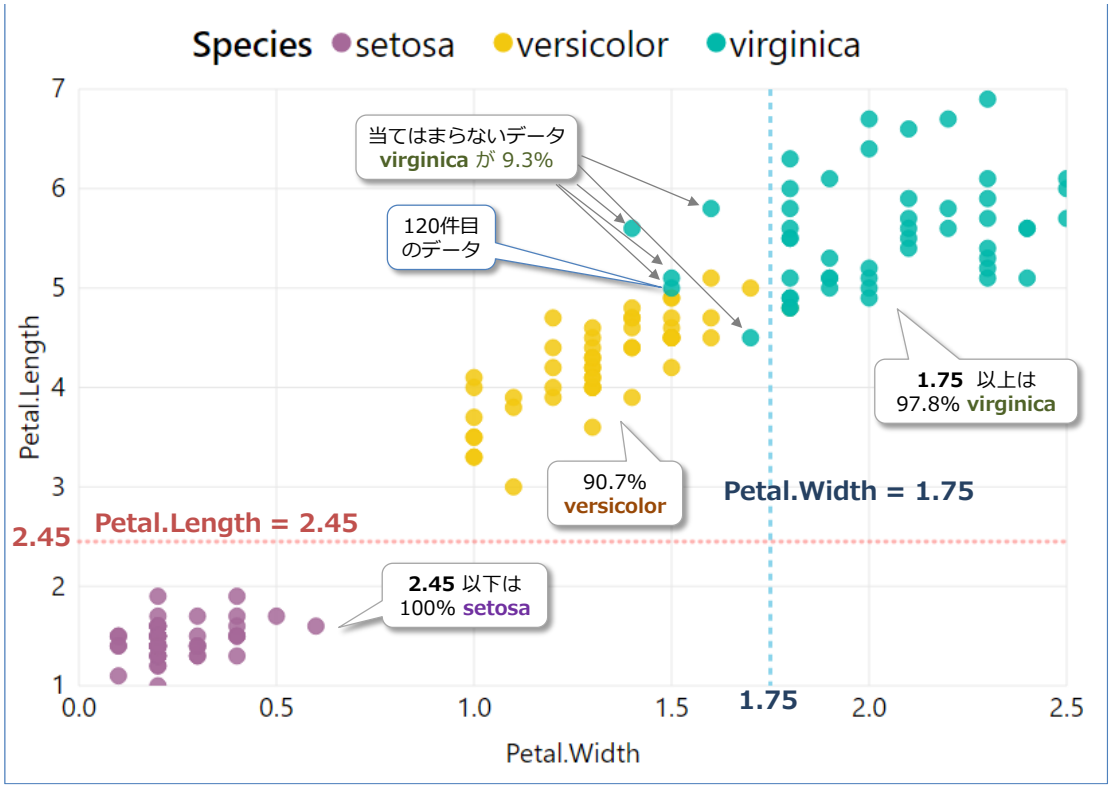
結果 メッセージ

	(列名なし)	(列名なし)	(列名なし)	(列名なし)	(列名なし)
119	7.7	2.6	6.9	2.3	virginica
120	6	2.2	5	1.5	virginica
121	6.9	3.2	5.7	2.3	virginica
122	5.6	2.8	4.9	2	virginica
123	7.7	2.8	6.7	2	virginica
124	6.3	2.7	4.9	1.8	virginica
125	6.7	3.3	5.7	2.1	virginica

120件目のデータ

120 件目の「**6.0、2.2、5.0、1.5**」という数値は、「**virginica**」で、**predict** 関数で予測した種類は「**versicolor**」なので、予測は失敗ということになります。

実際の 150 件のデータの分布（**Petal** の **Length** と **Width** に関する分布）は、前の項で確認しましたが、分布のグラフ（散布図）に、**決定木の分岐点**（**Petal.Length=2.45** と **Petal.Width=1.75**）を追加してみると、次のようになります。



Petal.Length < **2.45** では、**setosa** に **100%** 分類することができますが、**Petal.Width** < **1.75** では、**versicolor** がほとんど（**90.7%**）ですが、これに当てはまらないデータ（**virginica** のデータ）が **9.3%** 存在しています。この 9.3% に該当する 120 件目のデータは、Petal が「**5.0、1.5**」なので、Petal の特徴としては **versicolor** に似ているため、今回作成された決定木では判定できない（実際は virginica なのに versicolor と判定されてしまう）という形になります。

こうした似たような特性を持ったデータというのは、実際のデータではよくあることであり、

このようなデータをできる限り正確に予測していくために、いろいろなアルゴリズムが存在しています。例えば、後述の**ランダム フォレスト**や**サポート ベクター マシン**（SVM）、**ロジスティック回帰**、**ニューラル ネットワーク**などを利用すれば、予測の精度を上げていくことができます。また、データの特性に応じて、どのアルゴリズムが最適かはケース バイ ケースになるので、本格的に機械学習での予測に取り組む場合には、モデルを作成した後に、予測の精度（正解したのかどうか）を検証していくことが重要になります（より精度の高いものを採用して、性能面／モデル作成や予測スピードの実行時間も考慮しつつ、より良いサービスを提供していくためには重要な作業になります）。

このときによく行う手法が、実際のデータを**訓練データ**（トレーニング用データ）と**テスト データ**にランダムに分割して、訓練データでモデルを作成、テスト データでモデルの精度を検証する、といったことを行います（この作業はホールドアウト検証とも呼ばれています）。例えば、**150 件**の **iris** データであれば、ランダムに選択した **120 件**（**80%**）を訓練データとしてモデルを作成し、残りの **30 件**（**20%**）をテスト データとして利用して精度を検証したりします。こうしたデータの分割については後述します。

3. 次に、**Petal.Width** を「**1.8**」に変更して、実行してみます。

```
df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.8
                  ,Species = "dummy")
```

```
-- Petal.Width = 1.8 に変更
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
library(rpart)
model1 <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                ,data = iris)
# print(model1)

df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.8
                  ,Species = "dummy")

print(predict(model1, df1))
'
```

Petal.Length は 5.0

Petal.Width を 1.8 に変更

```
100 %
メッセージ
外部スクリプトからの STDOUT メッセージ:
  setosa versicolor virginica
1      0 0.02173913 0.9782609
```

Petal.Length が 2.45 以上で Petal.Width が 1.75 以上なので virginica と判定される

今度は、virginica と判定されたことを確認できます。

なお、ここの手順では、モデルの作成と予測を同じスクリプト内で実施しましたが、実際に業務で利用する場合には、別々のスクリプトで実行する場合がほとんどです（別々に実行するためには、モデルの保存が必要になりますが、これについても後述します）。

2.3 RevoScaleR の決定木（rxDTree）

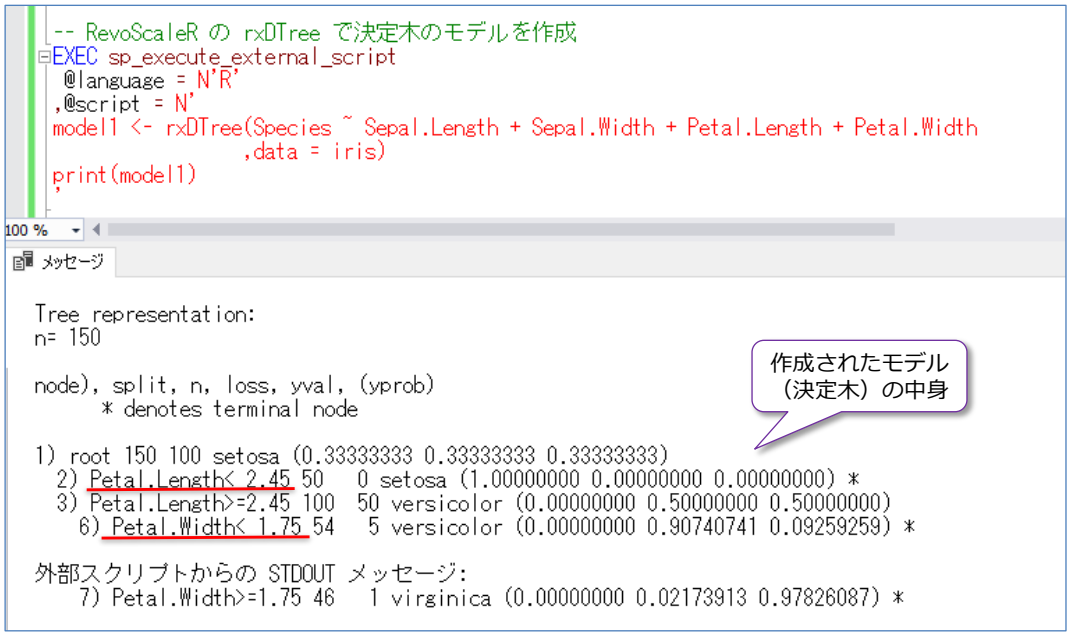
SQL Server 2017 の **Machine Learning Services** では、**RevoScaleR**（エンタープライズ対応の R）を利用できるのが大きなメリットです。RevoScaleR では、決定木のモデルを作成するために、「**rxDTree**」という関数が提供されています（Python の場合は **rx_dtree** という名前で提供）。使い方は、rpart 関数とほとんど同じです。

Let's Try

それでは、これも試してみましょう。

1. **rxDTree** 関数で決定木のモデルを作成するには、次のように記述します（**iris** データを利用）。

```
-- RevoScaleR の rxDTree で決定木のモデルを作成
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
model1 <- rxDTree(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                  ,data = iris)
print(model1)
'
```



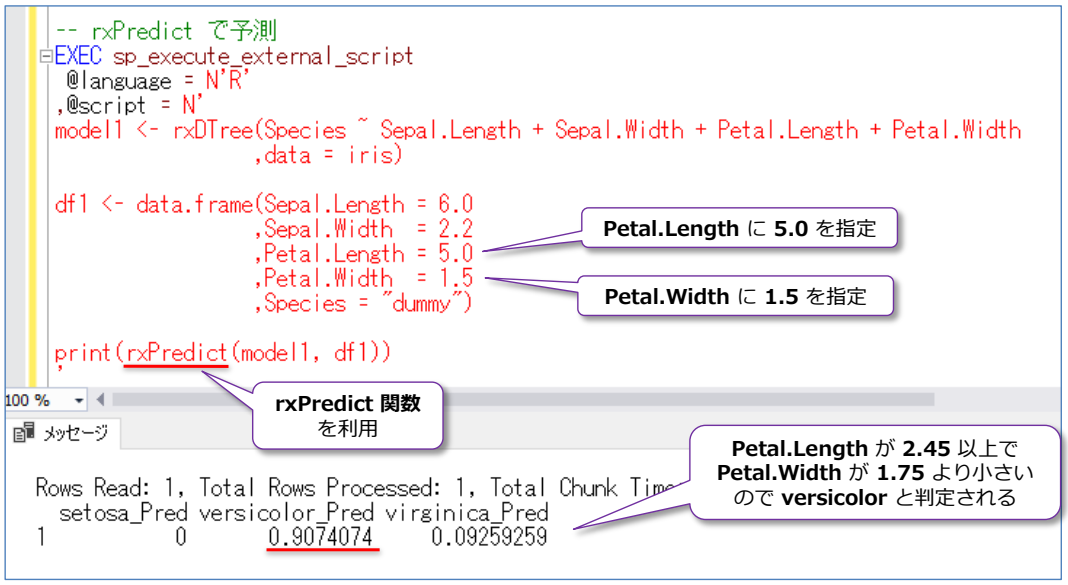
rpart 関数の場合と同様、種類（**Species**）を予測するために、「**~**」（チルダ）を入れて、**Sepal.Length** と **Sepal.Width**、**Petal.Length**、**Petal.Width** の 4 つの列を指定します。

作成されたモデル（**model1**）についても、**rpart** 関数の場合と同様、**Petal.Length** < **2.45** の分岐と、**Petal.Width** < **1.75** の分岐があることを確認できます。

2. 次に、予測を行ってみましょう。RevoScaleR で作成したモデルに対しては、「**rxPredict**」という関数を利用して予測を行います。使い方は、前掲の **predict** 関数の場合と同様で、次の

ように記述します。

```
-- rxPredict で予測
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
model1 <- rxDTree(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                  ,data = iris)

df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.5
                  ,Species = "dummy")

print(rxPredict(model1, df1))
'
```

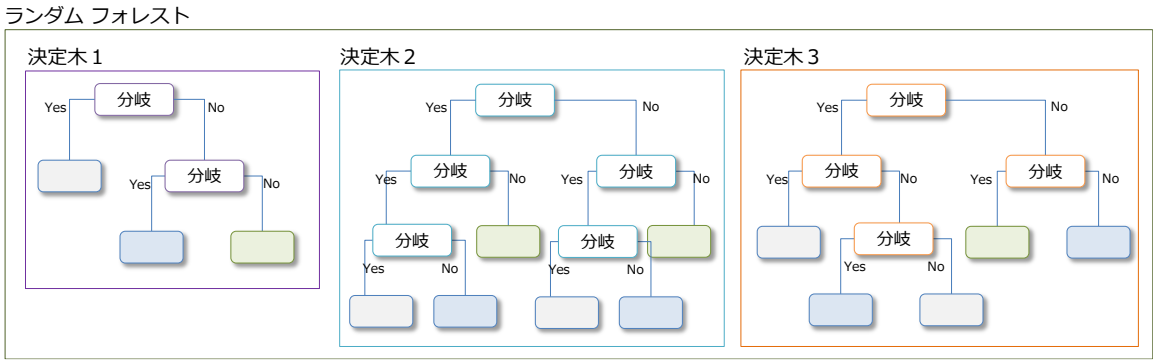


rxPredict 関数でも、第１引数にモデルを指定して、第２引数にデータ フレーム（予測を行いたいデータ）を指定します。

2.4 ランダム フォレスト（Random Forest）

次に、決定木よりも予測の精度が高い、**ランダム フォレスト**（**Random Forest**）を説明します。ランダム フォレストは、**Randomized Trees** と呼ばれることもあり、訓練データをランダムに抽出して、**複数の決定木**（Decision Tree）を作成するといった特徴があります。決定木（ツリー）がたくさん作られることから、**フォレスト**（Forest：森）というネーミングになっています。

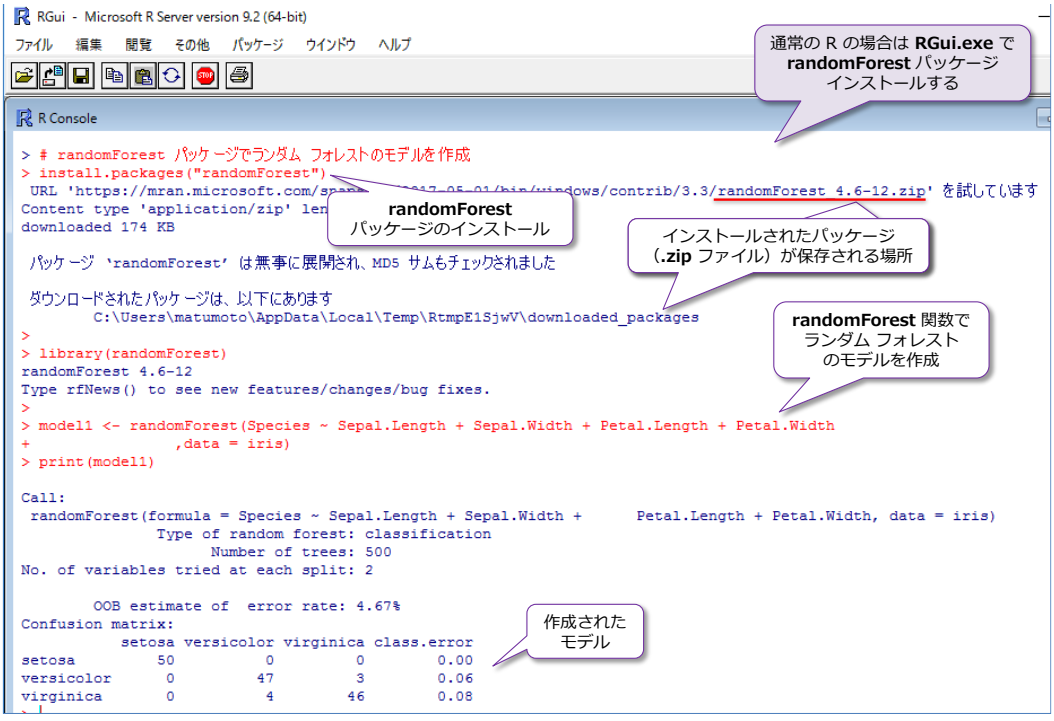
ランダム フォレスト（ランダム サンプリングで複数の決定木を作成）



ランダム フォレストでは、この複数の決定木をもとに**多数決**で予測を行うので、決定木よりも予測の精度が高くなることが多く、機械学習ではよく利用されるアルゴリズムです。

ランダム フォレストは、R 言語では「**randomForest**」パッケージや RevoScaleR の「**rxDForest**」、Python の場合は「**rx_dforest**」や scikit-learn ライブラリの「**RandomForest Classifier**」などを利用してモデルを作成することができます。

ここでは、まず R 言語での「**randomForest**」パッケージを利用する手順を説明しますが、通常の R 言語の場合には、次のように「**install.packages**」でパッケージをインストールしてから利用します。



これは **RGui.exe** ツールを利用して、通常の R スクリプト（SQL Server に統合された R スクリプトではなく、一般的な R スクリプト）を実行している例ですが、外部パッケージを利用する場合には、「**install.packages("外部パッケージ名")**」と記述して、外部パッケージをインストールしています。これによって、パッケージの **.zip** ファイル（**randomForest** の場合は、執筆時点では **randomForest_4.6-12.zip** という名前のファイル）がダウンロードおよびインストールされて、パッケージを利用できるようになります。

これに対して、SQL Server に統合された R（Machine Learning Services の R）を利用する場合には、外部パッケージは、別途 **.zip** ファイルをダウンロード、または前掲の「**install.packages**」でインストールされた .zip ファイルを利用して、**CREATE EXTERNAL LIBRARY** ステートメントを実行し、パッケージを登録しておく必要があります。

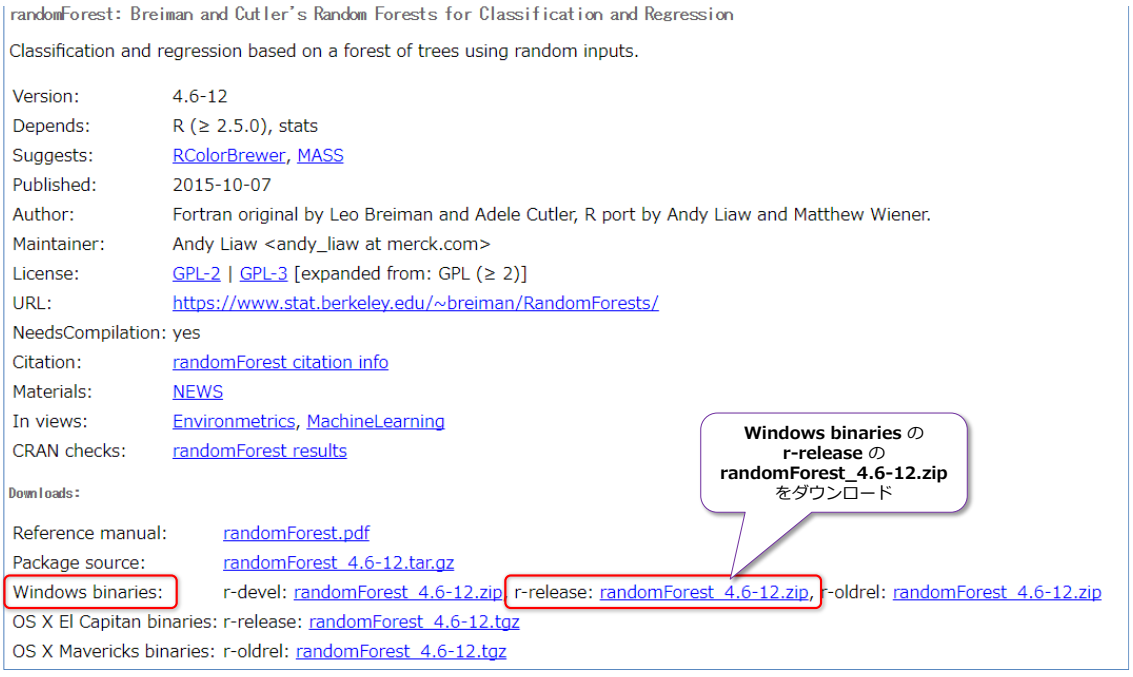
R で外部パッケージの利用（CREATE EXTERNAL LIBRARY）

Machine Learning Services の R では、外部パッケージを利用する場合には、次のように **CREATE EXTERNAL LIBRARY** ステートメントを実行して、パッケージを登録しておく必要があります。

```
-- 外部パッケージの登録
CREATE EXTERNAL LIBRARY 外部パッケージ名
FROM (CONTENT = '外部パッケージの zip ファイルへのパス')
WITH (LANGUAGE = 'R')
```

randomForest パッケージの場合は、次の URL から .zip ファイルをダウンロードできます。

randomForest
https://cran.r-project.org/web/packages/randomForest/index.html



この URL の「**Windows binaries**」セクションの「**r-release:**」にある「**randomForest_4.6-12.zip**」ファイルをクリックして、ダウンロードしておきます。

.zip ファイルのダウンロードが完了したら、.zip ファイルを任意のフォルダーに移動して、次のように **CREATE EXTERNAL LIBRARY** ステートメントを実行します（**C:¥temp** フォルダーを指定している部分は、皆さんの環境に合わせて移動したフォルダーに変更してください）。

```
-- randomForest パッケージの登録
CREATE EXTERNAL LIBRARY randomForest
FROM (CONTENT = 'C:¥temp¥randomForest_4.6-12.zip')
WITH (LANGUAGE = 'R')
```

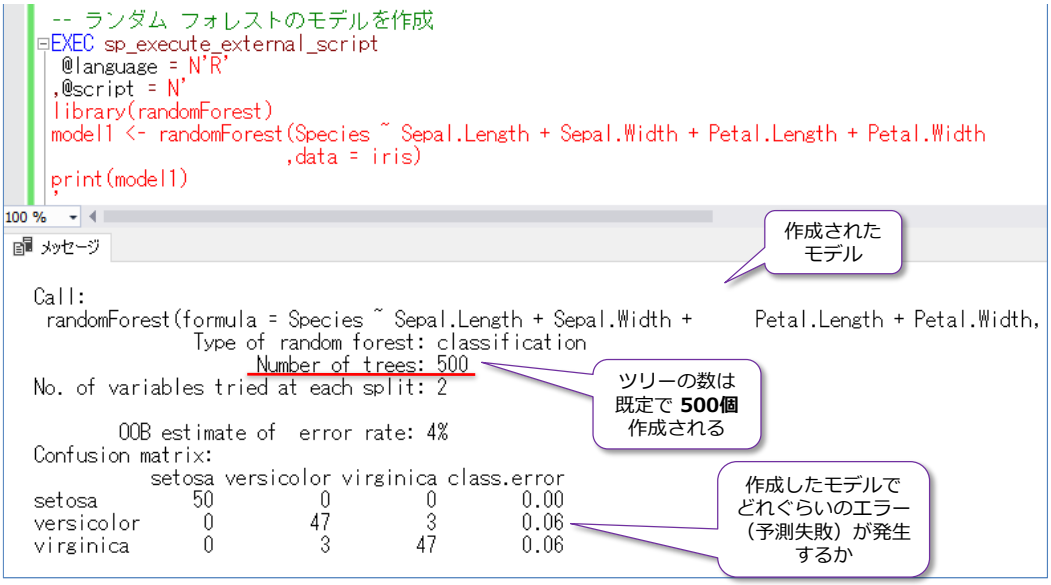
ランダム フォレストのモデルを作成

次に、ランダム フォレストのモデルを作成してみましょう。モデルの作成は、前掲の **rpart** や **rxDTree** 関数で決定木を作成した場合と、ほとんど同じです。

```
-- ランダム フォレストのモデルを作成
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
library(randomForest)
model1 <- randomForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                       ,data = iris)
print(model1)
'
```

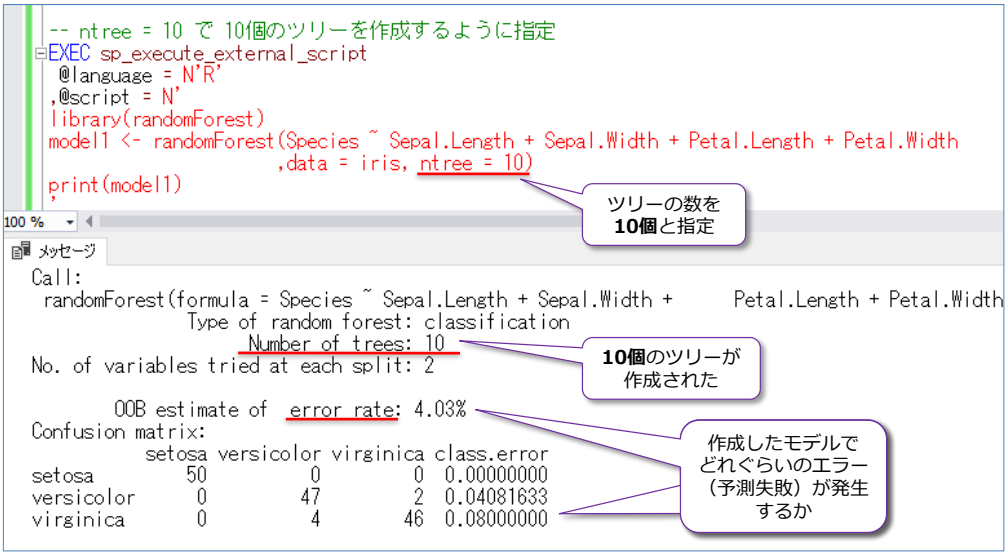
まず、**randomForest** パッケージを利用するために「**library(randomForest)**」を記述して、次に **randomForest** 関数でモデルを作成しています。引数の指定方法は、**rpart** や **rxDTree** 関数で決定木を作成した場合と同様、アヤメ（**iris**）の種類（**Species**）を予測するために、**Sepal.Length** と **Sepal.Width**、**Petal.Length**、**Petal.Width** の ４つを説明変数に指定しています。

作成されたモデル（**model1**）は、次のように確認できます。



randomForest 関数では、ツリー（決定木）の数を指定しなかった場合は **500 個**のツリーが作成されます。しかし、iris データは、150 件分のデータしかないので、500 個のツリーでは多すぎるので、関数の引数で、次のように「**ntree=ツリーの数**」を指定すれば、作成するツリーの数を変更することができます。

```
-- ntree = 10 で 10個のツリーを作成するように指定
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
library(randomForest)
model1 <- randomForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                       ,data = iris, ntree = 10)
print(model1)
'
```



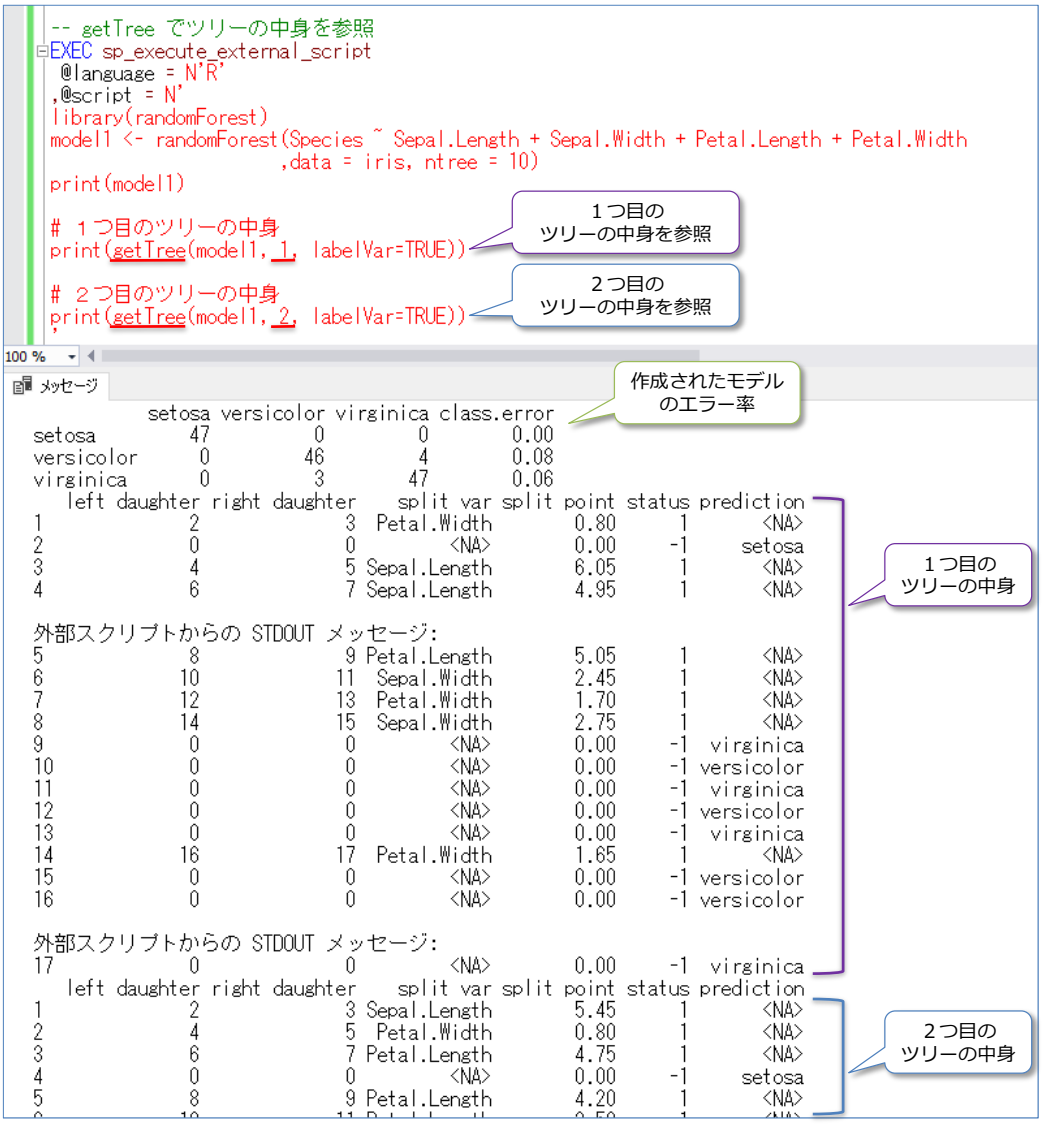
どのぐらいのツリー数を指定するのかは、**モデルの精度**（予測の正確さ、error rate：失敗率）と、**性能**（モデルの作成および予測にかかる実行時間、メモリ使用量）とのトレード オフになりますが、データの特性によっても、どのツリー数が最適かという答えが変わってくるので、他のアルゴリズ

ムも含めて、いろいろなパターンを検証しておくことが重要になります。

ランダム フォレスト内の決定木の中身

randomForest 関数で作成したランダム フォレストの場合は、**getTree** という関数を利用して、決定木（ツリー）の中身を参照することができます。これは、次のように利用できます。

```
library(randomForest)
model1 <- randomForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                       ,data = iris, ntree = 10)
print(model1)

# １つ目のツリーの中身
print(getTree(model1, 1, labelVar=TRUE))

# ２つ目のツリーの中身
print(getTree(model1, 2, labelVar=TRUE))
```



getTree 関数は、「**getTree(モデル名, ツリー番号, labelVar=TRUE)**」という形で利用するので、ツリー番号に **1** を指定すれば、1 つ目のツリーの中身を参照することができます。ランダム フォレストでは、データをランダムに抽出するので、どういったツリーが作成されるかは実行のたびに変わるのですが、上の 1 つ目のツリーを図で表現すると、次のようになります。

ランダム フォレスト内の決定木（ツリー）の例

	left daughter	right daughter	split var	split point	status	prediction
1	2	3	Petal.Width	0.80	1	<NA>
2	0	0	<NA>	0.00	-1	setosa
3	4	5	Sepal.Length	6.05	1	<NA>
4	6	7	Sepal.Length	4.95	1	<NA>
5	8	9	Petal.Length	5.05	1	<NA>
6	10	11	Sepal.Width	2.45	1	<NA>
7	12	13	Petal.Width	1.70	1	<NA>
8	14	15	Sepal.Width	2.75	1	<NA>
9	0	0	<NA>	0.00	-1	virginica
10	0	0	<NA>	0.00	-1	versicolor
11	0	0	<NA>	0.00	-1	virginica
12	0	0	<NA>	0.00	-1	versicolor
13	0	0	<NA>	0.00	-1	virginica
14	16	17	Petal.Width	1.65	1	<NA>
15	0	0	<NA>	0.00	-1	versicolor
16	0	0	<NA>	0.00	-1	versicolor
17	0	0	<NA>	0.00	-1	virginica

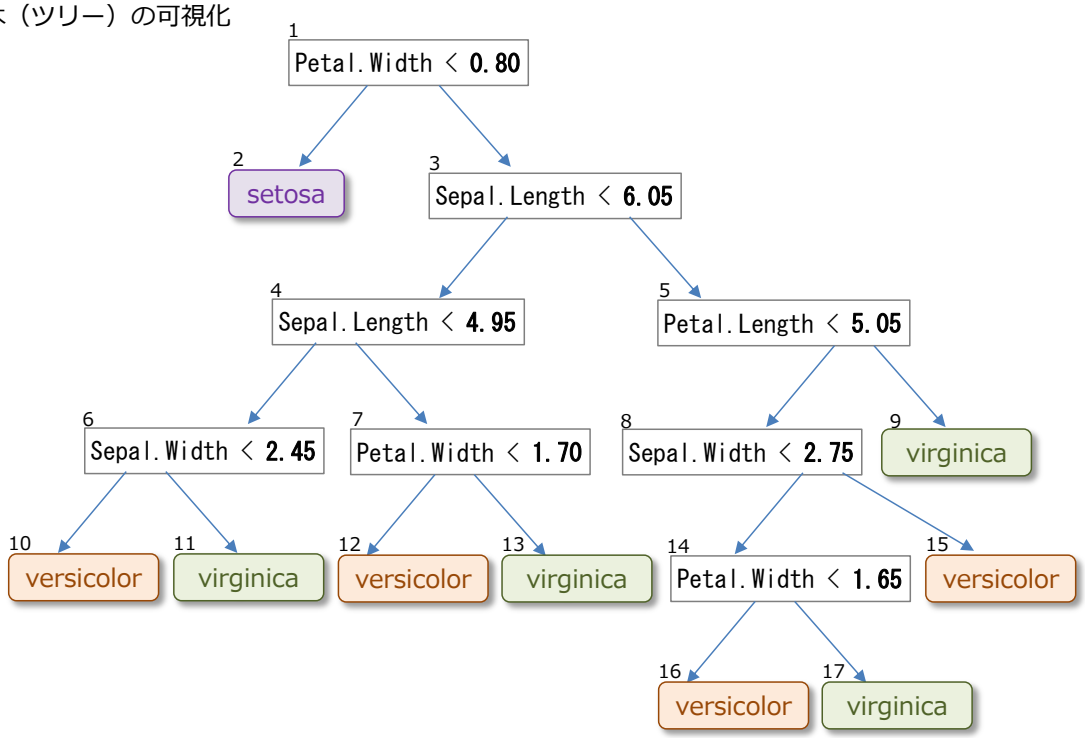
ノード ID

left daughter は 左下のノード ID。 0 の場合はリーフ

right daughter は 右下のノード ID。 0 の場合はリーフ

split point が分岐データ値

決定木（ツリー）の可視化



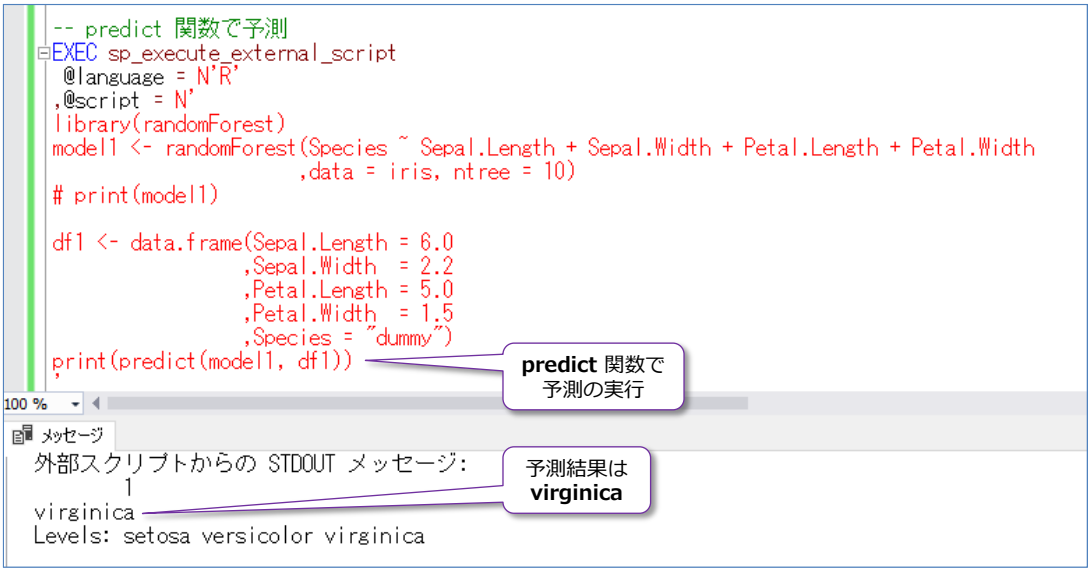
ランダム フォレストでは、このようにランダムに抽出したデータをもとに、複数の決定木（ツリー）を作成することで、予測の精度を高めています。

ランダム フォレストのモデルで予測（Predict）

次に、ランダム フォレストで作成したモデルを利用して、予測を行ってみましょう。予測は、**rpart** で決定木のモデルを作成したときと同様、**predict** 関数を利用します（利用方法も同じです）。

これも試してみましょう。

```
-- predict 関数で予測
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
library(randomForest)
model1 <- randomForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                       ,data = iris, ntree = 10)
# print(model1)

df1 <- data.frame(Sepal.Length = 6.0
                 ,Sepal.Width  = 2.2
                 ,Petal.Length = 5.0
                 ,Petal.Width  = 1.5
                 ,Species = "dummy")
print(predict(model1, df1))
'
```

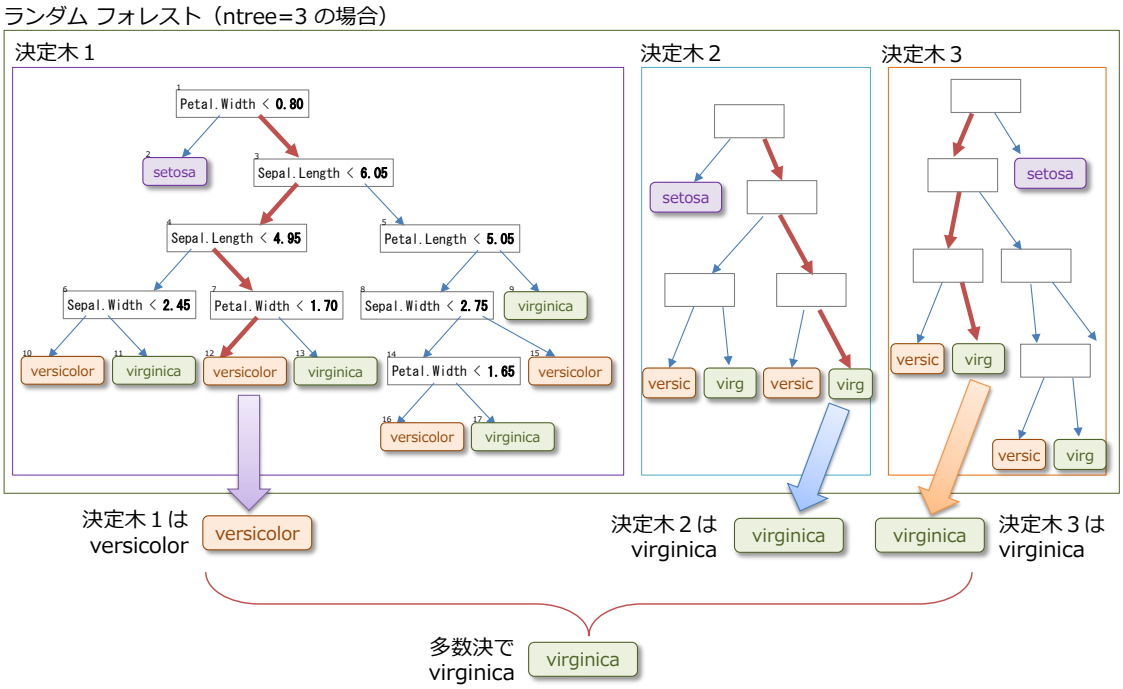


この例で指定した「**6.0、2.2、5.0、1.5**」という値は、実際の **iris** データの中にある、120 件目と同じもので、前掲の **rpart** での決定木では **predict** 関数で **versicolor** と判定されていたものです。正解は **virginica** なので、上の例では予測が正解しています。

ただし、ランダム フォレストでは、ツリーの数や、ランダム抽出によって、実行のたびにモデルが変わるので（後述のモデルの保存を行った場合は、保存したモデルを利用できますが）、**virginica** ではなく、**versicolor** と判定される場合（不正解の場合）もあります。「**6.0、2.2、5.0、1.5**」という値は、どの機械学習アルゴリズムを利用しても、**virginica** と **versicolor** のどちらにも予測される可能性がある微妙なデータになるので、判定が難しいものになっています。

なお、ランダム フォレストでの予測は、複数の決定木（ツリー）から多数決で行います。これは次のようなイメージになります。

ランダム フォレストでは複数の決定木から多数決で採用



ランダム フォレスト内に、どのぐらいのツリーを作成するのかは、前述したように、**モデルの精度**（予測の正確さ）と、**性能**（モデルの作成および予測にかかる実行時間、メモリ使用量）とのトレード オフになり、データの特性によっても、どのツリー数が最適かという答えが変わってきます。また、ここで使用した randomForest パッケージは、ツリー数を増やせば増やすだけ、メモリをその分消費していくので、搭載メモリが少ない場合には、ツリー数を多くしすぎると、メモリ不足で実行エラーになる場合もあります。

メモリ使用量は、データ量や説明変数で指定した列の数によっても変わってきますが、次に説明する **RevoScaleR** の **rxDForest** 関数を利用すれば、メモリ使用量を抑えることができます（少ないメモリでもランダム フォレストのモデルを作成できるように、大規模データ向けに実装されています）。

なお、SQL Server 2017 の Machine Learning Services は、既定では、最大でメモリ使用量の **20%** を利用するように設定されていますが、この変更方法については、4 章で説明します（リソース ガバナーで使用量を変更できます）。

2.5 RevoScaleR のランダム フォレスト（rxDForest）

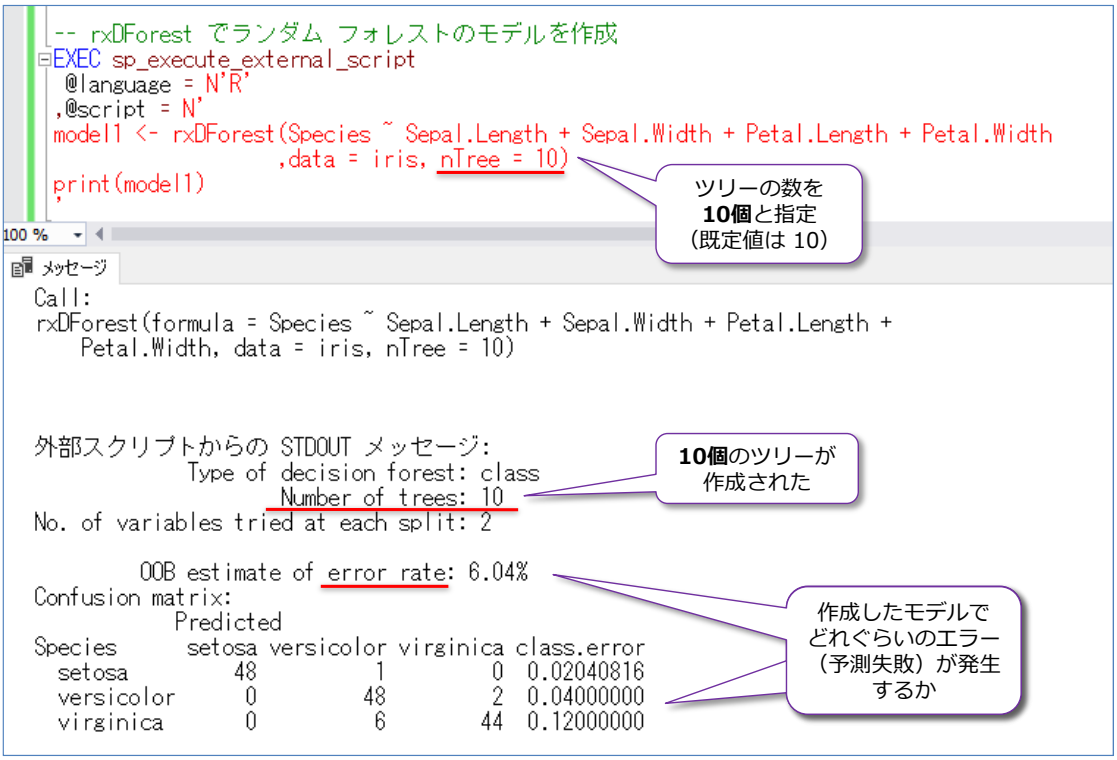
RevoScaleR では、「**rxDForest**」という関数でランダム フォレストのモデルを作成することができます（Python の場合は **rx_dforest** という名前で提供）。**rxDForest** 関数の使い方は、**randomForest** 関数の場合とほとんど同じです（**predict** 関数は、**rxPredict** 関数に変わります。Python の場合は **rx_predict** 関数）。

Let's Try

それでは、これも試してみましょう。

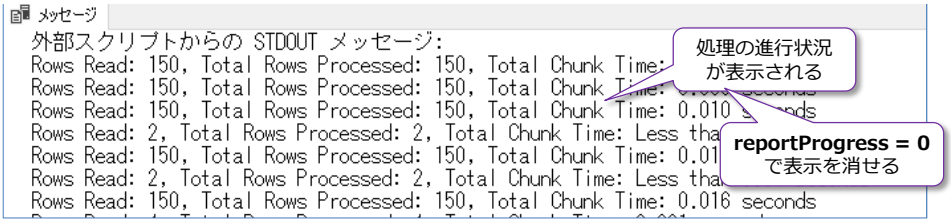
1. **rxDForest** 関数でランダム フォレストのモデルを作成するには、次のように記述します。

```
-- RevoScaleR の rxDTree で決定木のモデルを作成
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
model1 <- rxDForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                    ,data = iris, nTree = 10)
print(model1)
'
```



randomForest 関数を指定していた部分を **rxDForest 関数**に変更して、「**ntree = 10**」を「**nTree = 10**」に変更しています（**Tree** の **T** が大文字に変わります）。また、**randomForest** 関数でのツリー数は、既定で 500 個でしたが、**rxDForest 関数**の場合の既定値は 10 個です（このため、**nTree = 10** の場合は省略できます）。

なお、**RevoScaleR** の関数では、既定では「**Rows Read: ～**」のような進行状況メッセージが表示されます。

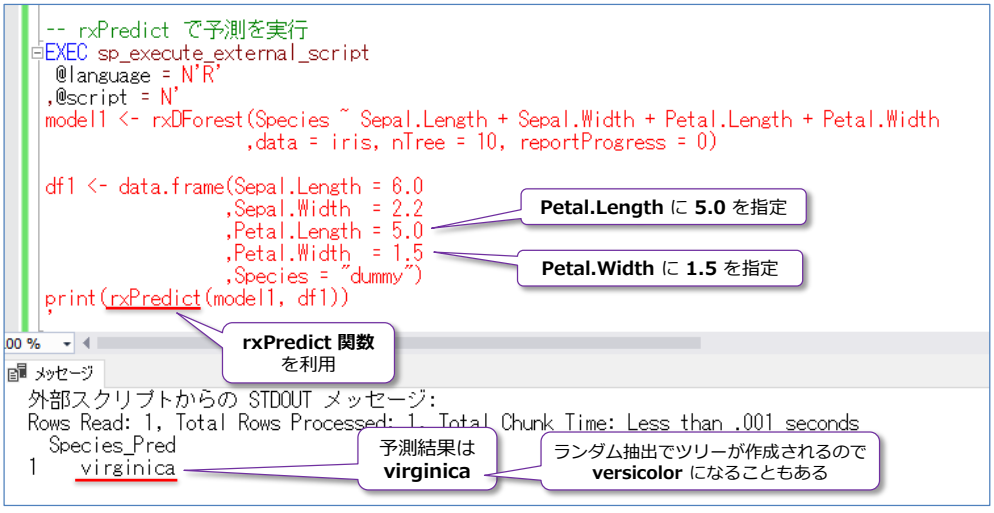


このメッセージを消したい場合には、関数の引数に「**reportProgress = 0**」を追加します。

```
model1 <- rxDForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                    ,data = iris, nTree = 10, reportProgress = 0)
print(model1)
```

2. 次に、予測を行ってみましょう。RevoScaleR で作成したモデルに対しては、**rxDTree** の場合と同様、「**rxPredict**」関数を利用して予測を行います。

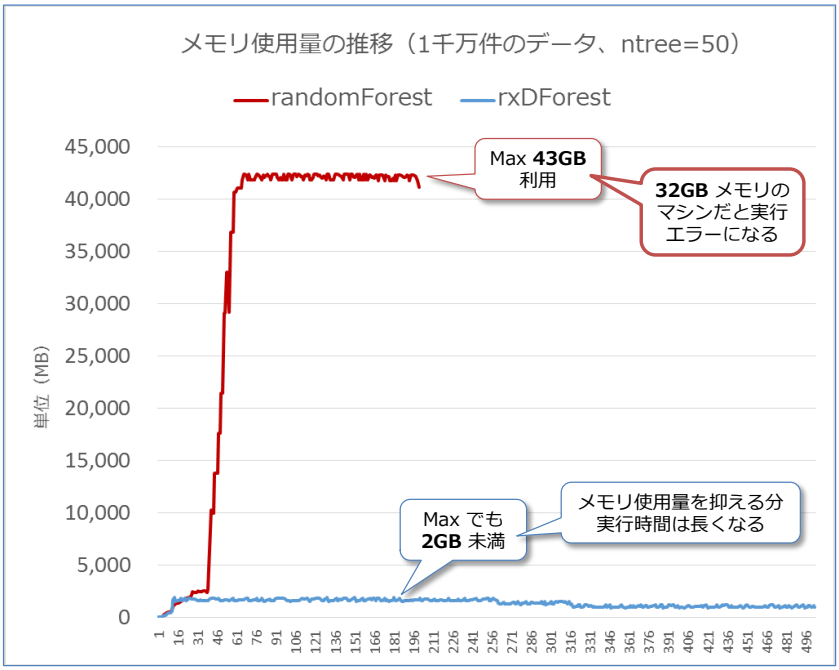
```
-- rxPredict で予測
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
model1 <- rxDForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                    ,data = iris, nTree = 10, reportProgress = 0)

df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.5
                  ,Species = "dummy")
print(rxPredict(model1, df1))
'
```



randomForest 関数での **predict** と同様、**virginica** または **versicolor** に判定されることを確認できると思います。

メモリ使用量の差（randomForest vs. rxDForest）

randomForest 関数と **rxDForest** 関数では、モデルを作成するときのメモリ使用量に大きな差が生まれます。次のグラフは、弊社環境で 1 千万件のデータに対して、ランダム フォレストのモデルを作成したときのメモリ使用量の推移です。



■ テスト内容
1千万件のデータ（テーブルサイズは約 600MB）に対してランダム フォレストのモデル作成を試みたときのメモリ使用量の推移。
説明変数は 5個、ntree=50

■ 使用したソフトウェア
・Windows Server 2016
・SQL Server 2017 Enterprise

■ テスト環境
CPU：Core i7-6700K 4コア
メモリ： 64GB
ストレージ：Crucial MX200 1TB

＊ ベンチマークの結果の公表は、使用許諾契約書で禁止されているので、X 軸の実行時間は相対値にしています。

randomForest 関数では、ツリー数を増やせば増やすほど、メモリをその分消費していくので、最大で **43GB** ものメモリを利用しています。このテストを行ったマシンは 64GB のメモリを搭載していたのでモデルを作成することができましたが、もし 32GB メモリのマシンを利用している場合には、メモリ不足で実行エラーになってしまいます。

これに対して、**RevoScaleR** の **rxDForest** 関数では、メモリ使用量は、最大でも **2GB** 未満で収まっているので、32GB メモリのマシンでも動作させることができます。その分、実行時間は長くなってしまいますが、**処理時のメモリ使用量を削減**することができます。RevoScaleR では実行するコンピューター リソースに応じてどのように動かすか（多くのリソースを使って速く動かすことも、限られたリソースのもとで制限して動かすことも可能）を柔軟にコントロールできるようになっています。

このように **RevoScaleR** は、大量のデータに対応するべく、メモリ使用量を抑えられるように実装されています。また、RevoScaleR では、さらにメモリ使用量を抑えて実行できるようにするために、読み取りデータ数（1 回のバッチ処理数）を指定して実行することもできます。詳しくは 4 章で説明しますが、例えば 1 千万件のデータの場合に、**rowsPerRead** というオプションで 100 万を設定すれば、100 万件ずつデータを処理できるようになります（メモリ使用量が 100 万件分で済むようになります）。データ件数を減らすと、その分実行時間は長くなってしまいますが、そこは大規模データに対応するためなので、メモリ使用量とのトレード オフになっています。

2.6 SQL Server のデータを利用してモデルを作成

ここまでは、R 言語に組み込まれた **iris** データセットを利用して、モデルを作成してきましたが、Machine Learning Services の R（SQL Server に統合された R）の一番の特徴であり、大きなメリットでもある、SQL Server 上のテーブルをデータ ソースにして、モデルを作成する手順を説明します。

iris データを SQL Server のテーブルに変換

まずは、R の iris データを SQL Server 上のテーブルに変換してみましょう。

```
-- データベースの作成。「mlTestDB」という名前で作成
CREATE DATABASE mlTestDB
go

-- テーブルの作成。「iris」という名前で作成
USE mlTestDB

CREATE TABLE iris
( [Sepal.Length] float
 ,[Sepal.Width]  float
 ,[Petal.Length] float
 ,[Petal.Width]  float
 ,Species varchar(100) )

-- R の iris データを iris テーブルに INSERT
INSERT INTO iris
 EXEC sp_execute_external_script
        @language = N'R'
       ,@script = N'OutputDataSet <- iris'

-- データの確認
SELECT * FROM iris
```

```
-- データの確認
SELECT * FROM iris
```

100 %

結果 メッセージ

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa

「**OutputDataSet <- iris**」という形で、**iris** データセットの中身を **OutputDataSet** という出

力変数に代入することで、結果をスクリプトの外に出力できるようになり、それを **INSERT** ステートメントでテーブルに格納しています。SQL Server の **INSERT** ステートメントでは、値を指定する **VALUES** の部分にストアド プロシージャ（**EXEC** で記述）を指定することができ、ストアド プロシージャの出力結果（**OutputDataSet**）をもとに、テーブルにデータを INSERT することができます。

なお、ここで作成した **iris** テーブルは、列名に「**Sepal.Length**」や「**Sepal.Width**」など、R の iris データセットの列名と同じものを利用しましたが、任意の列名でもかまいません。同じ名前にすることで、ここまで説明してきた R のスクリプトを、以降のスクリプトでもそのまま利用できるようになるので、ここでは同じ名前の列名にしています。

SQL Server のデータを利用してモデルを作成

次に、SQL Server 上の **iris** テーブルを利用して、モデルを作成してみましょう。SQL Server のデータを利用するには、**@input_data_1** パラメーターに **SELECT** ステートメントを指定して、次のように実行します（R スクリプトは、前の項で利用したものと 1 ヶ所違うだけです）。

```
-- @input_data_1 に SELECT ステートメントを指定して、data = InputDataSet に変更
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
model1 <- rxDForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                    ,data = InputDataSet, nTree = 10, reportProgress = 0)

df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.5
                  ,Species = "dummy")
print(rxPredict(model1, df1))
'
,@input_data_1 = N'SELECT * FROM iris'
```

```
-- @input_data_1 に SELECT ステートメントを指定
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
model1 <- rxDForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                    ,data = InputDataSet, nTree = 10, reportProgress = 0)

df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.5
                  ,Species = "dummy")
print(rxPredict(model1, df1))
'
,@input_data_1 = N'SELECT * FROM iris'
```

2 data = InputDataSet に変更

1 SELECT ステートメントで iris テーブルを指定

```
100 %
メッセージ
外部スクリプトからの STDOUT メッセージ:
Rows Read: 1, Total Rows Processed: 1, Total Chunk Time: 0.001 seconds
  Species_Pred
1    virginica
```

予測結果

前の項との違いは、**rxDForest** 関数の引数で「**data=iris**」を指定していた部分を「**data=InputDataSet**」に変更しているだけです。**@input_data_1** パラメーターで取得した結果は、データ フレーム（**data.frame**）形式になっていて、既定では **InputDataSet** という変数名で利用することができるので、このように利用できます。

rxPredict 関数での予測の結果は、前の項と同様、**virginica** または **versicolor** になります。

InputDataSet の名前を変更したい場合（@input_data_1_name）

InputDataSet という変数名は、変更することもできます。これを行うには、次のように **@input_data_1_name** パラメーターに任意の名前を設定します。

```
-- @input_data_1_name に名前を指定
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
model1 <- rxDForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                   ,data = iris_table, nTree = 10, reportProgress = 0)

df1 <- data.frame(Sepal.Length = 6.0
                  ,Sepal.Width  = 2.2
                  ,Petal.Length = 5.0
                  ,Petal.Width  = 1.5
                  ,Species = "dummy")
print(rxPredict(model1, df1))
'
,@input_data_1 = N'SELECT * FROM iris'
,@input_data_1_name = N'iris_table'
```

ここでは、「**@input_data_1_name = N'iris_table'**」のように「**iris_table**」という名前を設定しているので、R スクリプト内では「**data=iris_table**」と指定することができます。

以上のように、SQL Server 2017 の Machine Learning Services では、SQL Server のデータを簡単に R スクリプトあるいは後述の Python スクリプトで利用することができるのが大きな特徴であり、大きなメリットです（1 章で説明したように、データの取り込みスピードについても抜群です）。

2.7 Python を利用する場合のモデル作成と予測

ここまでは、**R** 言語を利用して、決定木やランダム フォレストを試してきましたが、次に **Python** 言語（SQL Server に統合された Python）を利用する方法を説明します。

Python を利用する場合でも、**Revoscalepy**（**RevoScaleR** の Python 版）の関数を利用すれば R の場合とほとんど同じように利用できます。R での **rxDTree** は **rx_dtree**、**rxDForest** は、**rx_dforest** として提供されています。Python の場合は、関数名や引数名はすべて小文字になって、R で大文字だった部分には _ が入るのが基本的なネーミング ルールです。

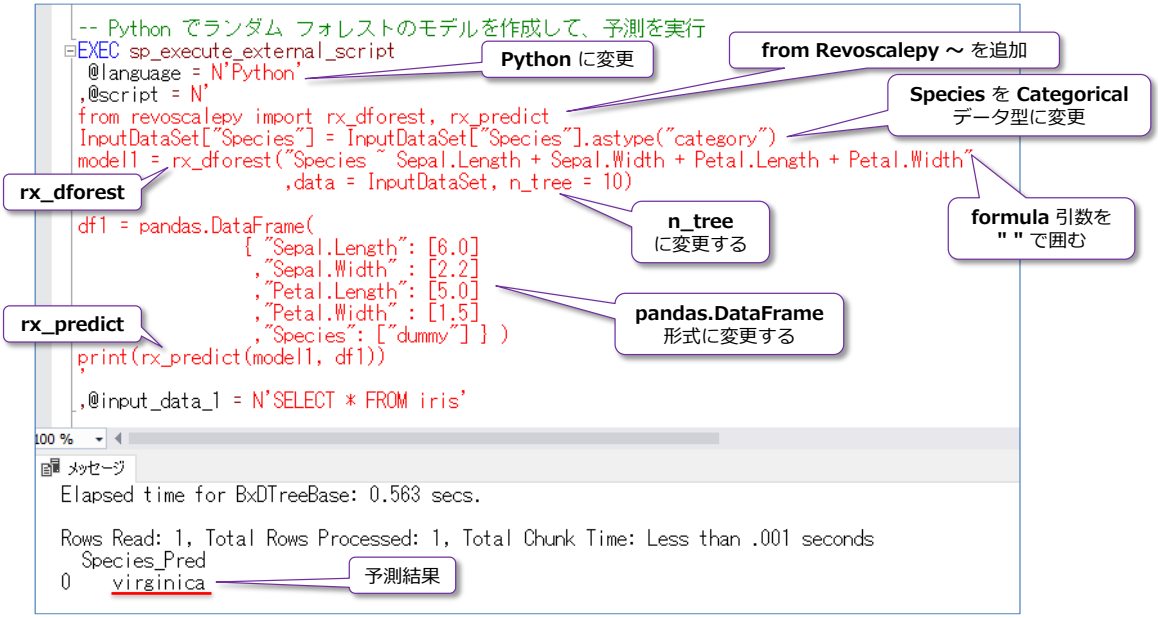
ここまでの R との主な変更点は、次のようになります。

- **@language** を **N'R'** から **N'Python'** に変更する
- 値の代入を「**<-**」から「**=**」に変更する
- **rxDForest** を **rx_dforest**、**rxPredict** を **rx_predict** に変更する
- **formula** 引数（目的変数と説明変数を指定している部分）を **" "**（二重引用符）で囲む
- **nTree** 引数を **n_tree** に変更する（T を小文字にして、_ を間に入れる）
- R での **data.frame** を **pandas.DataFrame** 形式に変更する
- rx_ 系の関数を利用するために **import** を記述する
- Species（アヤメの種類）列を **Categorical** データ型に変更する

Let's Try

それでは、これも試してみましょう。次のように記述して、**rx_dforest** 関数でランダム フォレストのモデルを作成して、**rx_predict** 関数で予測を実行してみます。

```
-- Python でランダム フォレストのモデルを作成して、予測を実行
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from revoscalepy import rx_dforest, rx_predict
InputDataSet["Species"] = InputDataSet["Species"].astype("category")
model1 = rx_dforest("Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width"
                    ,data = InputDataSet, n_tree = 10)

df1 = pandas.DataFrame(
                { "Sepal.Length": [6.0]
                 ,"Sepal.Width" : [2.2]
                 ,"Petal.Length": [5.0]
                 ,"Petal.Width" : [1.5]
                 ,"Species": ["dummy"] } )
print(rx_predict(model1, df1))
'
,@input_data_1 = N'SELECT * FROM iris'
```



Python を利用するには、**@language** を **N'R'** から **N'Python'** に変更します。Python スクリプトでは、R と同様、大文字と小文字を区別するので、R との微妙な違いに気を付けつつ、スクリプトを記述してください（予測結果が表示されずにエラーが表示される場合は、大文字と小文字のチェックをしてみてください）。

Python スクリプトの 1 行目では、「**from revoscalepy import rx_dforest, rx_predict**」と記述していますが、Python の場合は **rx_dforest** および **rx_predict** 関数を利用するために、この記述（**import**）が必須になります。

2 行目の「**InputDataSet["Species"]=～**」では、SQL Server 上の iris テーブルから取得した **Species** 列（アヤメの種類）を「**InputDataSet["Species"].astype("category")**」に置換しています。**astype("category")** と指定することで、**Categorical** データ型に変更しています。Python での **rx_dforest** 関数の目的変数（予測したい対象）には、**Categorical** データ型の列を指定する必要があるので、このように変換しています。

rx_dforest 関数の **formula** 引数（目的変数と説明変数を指定している部分）は、**" "**（二重引用符）で囲む必要があるので、「**"Species ~ Sepal.Length + Sepal.Width + ～"**」のように変更しています。また、**nTree** 引数は、**n_tree** に変更することに注意してください（T を小文字にして、_ を間に入れます）。

予測のためのデータは、R では **data.frame** を利用してデータ フレームを作成していましたが、Python では **pandas.DataFrame** を利用します。通常の Python スクリプトでは、**pandas** ライブラリは「**import pandas**」と記述して利用するのですが、**sp_execute_external_script** 内では、「**import pandas**」を記述するとスクリプト エラーになってしまうので注意してください。pandas は、内部的に「**import pandas**」が実行された状態になっていて、「**pandas**」という名前でそのまま利用することができるので、「**pandas.DataFrame**」と記述しています。

pandas.DataFrame では、次のようにデータ フレームを作成しています。

```
df1 = pandas.DataFrame(
                { "Sepal.Length": [6.0]
                 ,"Sepal.Width" : [2.2]
                 ,"Petal.Length": [5.0]
                 ,"Petal.Width" : [1.5]
                 ,"Species": ["dummy"] } )
```

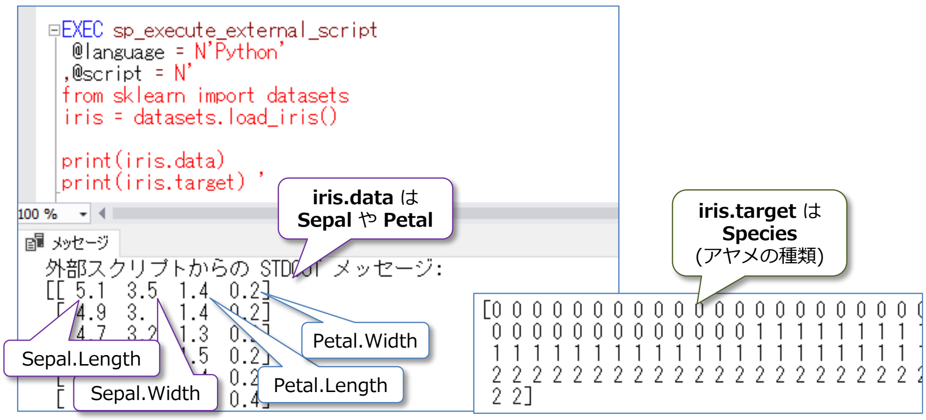
R では、「**data.frame(列名 1=値, 列名 2=値, …)**」という形で利用していましたが、Python の場合は「**pandas.DataFrame({ "列名 1": [値], "列名 2": [値], … })**」という形で利用する必要があります。全体を **{ }**（中カッコ）で囲んで、列名は**二重引用符**で囲み、値は **[ ]**（大カッコ）で囲みます。また、値の指定は「**=**」ではなく「**:**」（コロン）を利用することにも注意してください。このあたり、少しでも間違えるとスクリプト エラーになってしまうので、もしエラーになる場合は、正確に入力できているかどうかを確認してみてください（あるいは、サンプル スクリプトにスクリプト例を記載しているので、それをコピー＆ペーストして、実行してみてください）。

以上のように、Python を利用しても、R と同じように機械学習を行うことができます。

参考情報： scikit-learn（sklearn）の iris データを利用する場合

Python の機械学習では、「**scikit-learn**」（**sklearn**）というライブラリがよく利用されています。このライブラリには、**iris** データが付属していて、次のように **datasets.load_iris** メソッドで取得することができます。

```
-- sklearn の iris データの中身
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from sklearn import datasets
iris = datasets.load_iris()

# iris.data は Sepal や Petal
print(iris.data)

# iris.target は Species（種類を数値化したもの 0 はsetosa、1 はversicolor、2 はvirginica）
print(iris.target) '
```



load_iris メソッドでデータを取得すると、**iris.data** には **Sepal.Length** と **Sepal.Width**、**Petal.Length**、**Petal.Width** の 4 つの列、**iris.target** には **Species** 列が格納されています。**Species** 列は、アヤメの種類を数値化したもので、**0** は **setosa**、**1** は **versicolor**、**2** は **virginica** になっています。

この **iris** データを利用して、**rx_dforest** 関数でランダム フォレストのモデルを作成する場合には、次のように **iris.data** と **iris.target** を **pandas.DataFrame** 形式に変更して利用するようにします（**df_input** までが新しい記述で、**from revoscalepy** 以降は、今までのスクリプトと 1 ヶ所違うだけです。**data=InputDataSet** を **data=df_input** に変更しているのみ）。

```
-- sklearn の iris データで rx_dforest（ランダム フォレスト）
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from sklearn import datasets
iris = datasets.load_iris()

# iris.data を pandas.DataFrame に変換
df_data = pandas.DataFrame(iris.data
            ,columns=["Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width"])

# iris.target を pandas.DataFrame に変換
df_target = pandas.DataFrame(iris.target
            ,columns=["Species"])

# df_data と df_target を concat（連結、Join）
df_input = pandas.concat([df_data, df_target], axis=1)

# rx_dforest でランダム フォレストのモデルを作成
from revoscalepy import rx_dforest, rx_predict
model1 = rx_dforest("Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width"
                    ,data = df_input, n_tree = 10)

# 予測
df1 = pandas.DataFrame(
                { "Sepal.Length": [6.0]
                ,"Sepal.Width" : [2.2]
                ,"Petal.Length": [5.0]
                ,"Petal.Width" : [1.5]
                ,"Species": ["dummy"] } )
print(rx_predict(model1, df1))
'
```

「**df_data =**」では、**iris.data** を **pandas.DataFrame** 形式に変更していますが（**df_data** は変数名です）、「**pandas.DataFrame(iris.data, columns=["Sepal.Length", "～", …**」と記述することで、列名を **Sepal.Length** や **Sepal.Width** に設定することができます。列名は、任意のものを利用できるのですが、変更する場合は、**rx_dforest** の **formula** 引数で説明変数を指定している部分の列名も変更する必要があります。

「**df_target =**」では、**iris.target** を **pandas.DataFrame** 形式に変更して、列名を **Species**

に設定しています。

「df_input =」では、「**pandas.concat([df_data, df_target], axis=1)**」と記述することで、**df_data** と **df_target** を連結（concat）して、1 つのデータ フレームにすることができます。連結したデータは、次のような形式になります。

df_data と df_target を連結した df_input

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
5	5.4	3.9	1.7	0.4	0
6	4.6	3.4	1.4	0.3	0
7	5.0	3.4	1.5	0.2	0
8	4.4	2.9		0.2	0
9	4.9	3.1		0.1	0
10	5.4	3.7	1.5	0.2	0
11	4.8	3.4	1.6	0.2	0

df_data

df_target

この **df_input** を **rx_dforest** 関数の引数で「**data=df_input**」と与えれば、これまでの手順と同じようにランダム フォレストのモデルを作成することができます。

予測の結果は、次のとおりです。

```
-- sklearn の iris データで rx_dforest (ランダム フォレスト)
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from sklearn import datasets
iris = datasets.load_iris()

# iris.data を pandas.DataFrame に変換
df_data = pandas.DataFrame(iris.data
            ,columns=["Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width"])

# iris.target を pandas.DataFrame に変換
df_target = pandas.DataFrame(iris.target
            ,columns=["Species"])

# df_data と df_target を concat (連結、Join)
df_input = pandas.concat([df_data, df_target], axis=1)

# 連結したデータの確認
print(df_input)

# rx_dforest でランダム フォレストのモデルを作成
from revoscalepy import rx_dforest, rx_predict
model1 = rx_dforest("Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width"
                    ,data = df_input, n_tree = 10)

# 予測
df1 = pandas.DataFrame(
                { "Sepal.Length": [6.0]
                 ,"Sepal.Width" : [2.2]
                 ,"Petal.Length": [5.0]
                 ,"Petal.Width" : [1.5]
                 ,"Species": ["dummy"] } )
print(rx_predict(model1, df1))
'
```

df_input に変更

100 %

メッセージ

```
Rows Read: 1, Total Rows Processed: 1, Total
   Species_Pred
0      1.604826
```

予測結果

0 は setosa、1 は versicolor、2 は virginica なので 1.6 は virginica と判定されていることが分かる

予測結果は「**1.6～**」など、数値が表示されますが、Species 列の 0 は setosa、1 は versicolor、2 は virginica なので、1.6 であれば virginica と判定されていることが分かります。

2.8 モデルの保存とネイティブ スコアリング（PREDICT）

次に、作成したモデルを SQL Server のテーブルに保存して、それを利用して予測（Predict）を行う（予測を別途実行する形に変更する）方法を説明します。

モデルを保存するには、Python または R スクリプト内で作成したモデルを、スクリプトの外に出力する必要がありますが、これは **sp_execute_external_script** の **@params** パラメーターを利用して行うことができます。大まかな利用イメージは、次のようになります。

```
-- 出力結果を受け取るための変数を事前に定義
DECLARE @出力先 データ型

-- sp_execute_external_script の実行。@変数名 を出力する。
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
   :
# 出力したい処理結果（モデルetc）を変数に代入。変数は @params で定義しておく必要あり
変数名 = 処理結果
'
,@params = N'@変数名 データ型 OUTPUT'
,@変数名 = @出力先 OUTPUT
```

@params パラメーターで**変数名**と**データ型**を定義して、これに **OUTPUT** キーワードを付けておきます（変数名には **@** を付けておく必要があります）。この変数名は、スクリプト内で、@ を付けずに利用することができ、出力したい値を代入しておくことで、スクリプトの外に出力することができます。

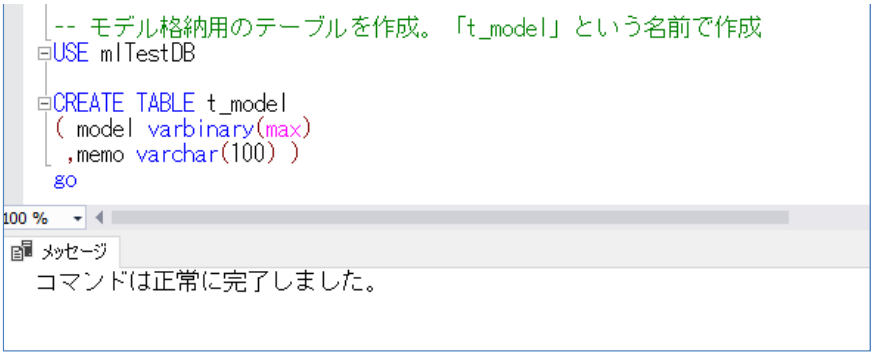
出力した変数は、「**@変数名 = @出力先 OUTPUT**」で、事前に定義した（スクリプトの外側で、最初に定義した）出力先用の変数「**@出力先**」に格納することができます。

また、モデルを出力するにあたっては、**rx_serialize_model**（R の場合は **rxSerializeModel**）関数を利用して、モデルをシリアライズ化しておくことで、ネイティブ スコアリング（Transact-SQL ステートメントの **PREDICT** 関数を利用して予測の実行）ができるようになります。

Let's Try

それでは、これも試してみましょう。まずは、モデルを保存するためのテーブルを SQL Server 上に作成します。

```
-- モデル格納用のテーブルを作成。「t_model」という名前で作成
USE mlTestDB

CREATE TABLE t_model
( model varbinary(max)
 ,memo varchar(100) )
```



テーブル名は「**t_model**」として、「**model**」列にモデル データ（バイナリ形式）を格納できるようにするために **varbinary(max)** データ型を利用しています。また、このテーブルには、複数のモデルを格納できるようにするために、「**memo**」列（データ型は **varchar(100)**）を追加して、モデルを識別するためのメモ書き用の列として利用します。

次に、Python を利用して、**rx_dforest** 関数で作成したモデルを、この「**t_model**」テーブルに格納しますが、モデルをシリアライズ化するために **rx_serialize_model** 関数を利用したり、結果を受け取るための変数を事前に定義したり、**@params** で変数を定義したりします（以下のように実行します）。

```
-- 出力結果を受け取るための変数を事前に定義
DECLARE @output_model varbinary(max)

-- モデルをテーブルに INSERT
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
# rx_serialize_model のインポートを追加
from revoscalepy import rx_dforest, rx_serialize_model

InputDataSet["Species"] = InputDataSet["Species"].astype("category")
model1 = rx_dforest("Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width"
                    ,data = InputDataSet, n_tree = 10)

# モデルを rx_serialize_model 関数でシリアライズ化。output_model は @params で定義
output_model = rx_serialize_model(model1, realtime_scoring_only = True)
'
,@input_data_1 = N'SELECT * FROM iris'
,@params = N'@output_model varbinary(max) OUTPUT'
,@output_model = @output_model OUTPUT

INSERT INTO t_model VALUES(@output_model, 'rx_dforest')
```

sp_execute_external_script の **@params** では、「**@output_model varbinary(max) OUTPUT**」という形で、@ 付きの変数として「**@output_model**」という名前、データ型を **varbinary(max)**、**OUTPUT** キーワードを付けることで、スクリプト内で「**output_model**」という名前の出力変数として利用できます（スクリプト内では @ なしで利用します）。

この「**output_model**」変数には、**rx_serialize_model** 関数でモデル（**model1**）をシリアラ

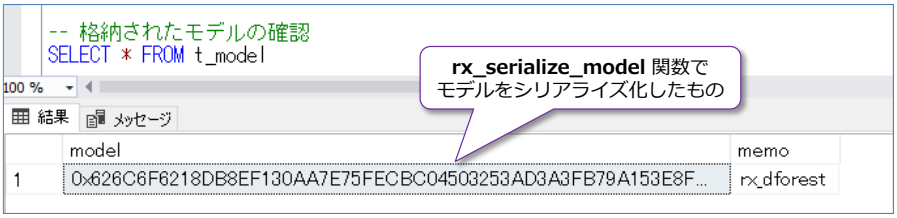
イズ化したものを代入しています。**rx_serialize_model** 関数では、第１引数にモデル名、第２引数には「**realtime_scoring_only = True**」を指定する必要があります（後述の **PREDICT** 関数を利用した予測でのみに利用するという意味の True 指定になります）。

スクリプト内で「**output_model**」変数に格納したモデルは、「**@output_model = @output_model OUTPUT**」によって、事前に定義した（スクリプトの外で **DECLARE** で最初に定義した）**@output_model** 変数（Transact-SQL の変数）に出力することができます。

これを「**INSERT INTO t_model VALUES(@output_model, 'rx_dforest')**」のように **VALUES** で指定することで、**t_model** テーブルの **model** 列に INSERT することができます。**memo** 列は、モデルを識別するための列になるので「**'rx_dforest'**」など任意の文字列を指定しておきます。

モデルの INSERT が完了したら、SELECT ステートメントを実行して、モデルが格納されたことを確認しておきます。

```
-- 格納されたモデルの確認
SELECT * FROM t_model
```



ネイティブ スコアリング（Transact-SQL の PREDICT 関数）

次に、保存したモデルを利用して、Transact-SQL ステートメントの **PREDICT** 関数を利用して、予測を実行してみましょう。

まずは、PREDICT 関数を実行する前に、予測のためのデータを「**test_data**」という名前のテーブルに格納しておきます（**SELECT .. INTO** で **iris** テーブルと同じ構造のテーブルを作成して、**INSERT** ステートメントで **２件**のデータを格納しておきます）。

```
-- テスト用のデータをテーブルに INSERT。「test_data」という名前で作成
USE mlTestDB
SELECT * INTO test_data FROM iris WHERE 1=2

INSERT INTO test_data
 VALUES (6.0, 2.2, 5.0, 1.5, 'dummy')
       ,(4.0, 2.0, 2.0, 0.7, 'dummy')

-- データの確認
SELECT * FROM test_data
```

```
-- テスト用のデータをテーブルに INSERT。「test_data」という名前で作成
USE mlTestDB
SELECT * INTO test_data FROM iris WHERE 1=2

INSERT INTO test_data
 VALUES (6.0, 2.2, 5.0, 1.5, 'dummy')
       ,(4.0, 2.0, 2.0, 0.7, 'dummy')

-- データの確認
SELECT * FROM test_data
```

100 %

結果 メッセージ

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	6	2.2	5	1.5	dummy
2	4	2	2	0.7	dummy

次に、Transact-SQL ステートメントの **PREDICT** 関数を利用して、予測を実行します。

```
-- PREDICT 関数を利用してネイティブ スコアリング
-- t_model テーブルの model 列を @model 変数に格納
DECLARE @model varbinary(max)
SELECT @model = model FROM t_model WHERE memo = 'rx_dforest'

SELECT *
 FROM PREDICT ( MODEL = @model, DATA = test_data )
 WITH ( setosa_prob float
       ,versicolor_prob float
       ,virginica_prob float
       ,Species_Pred nvarchar(200) ) AS p
```

```
-- ネイティブ スコアリング
DECLARE @model varbinary(max)
SELECT @model = model FROM t_model WHERE memo = 'rx_dforest'

SELECT *
 FROM PREDICT ( MODEL = @model, DATA = test_data )
 WITH ( setosa_prob float
       ,versicolor_prob float
       ,virginica_prob float
       ,Species_Pred nvarchar(200) ) AS p
```

予測結果

test_data テーブル のデータ

100 %

結果 メッセージ

	setosa_prob	versicolor_prob	virginica_prob	Species_Pred	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	0.00838382888390278	0.703641619532833	0.287974551583264	versicolor	6	2.2	5	1.5
2	0.912206572769953	0.0877934272300469	0	setosa	4	2	2	0.7

PREDICT 関数では「**MODEL=**」でシリアライズ化したモデル（**t_model** テーブルの **model** 列から取得したモデル）を指定して、「**DATA=**」で予測したいデータが格納されているテーブル名やビュー名、ストアド プロシージャ名（SELECT 結果を返すもの）などを指定するので、**test_data** テーブルを与えています。

WITH 句では、「**setosa_prob float, versicolor_prob float, ～**」などを指定していますが、この指定方法は、ちょっと難しくて、ランダム フォレスト（rx_dforest）の場合には、目的変数で指定した **Species** の種類（setosa、versicolor、virginica）に「**_prob**」が付いた名前の列名を指定する必要があり、それぞれの種類の可能性を出力するために利用します（例えば、1 件目は **versicolor_prob** が **0.703～** 、**virginica_prob** が **0.287～** なので、**versicolor** に判定されて、2 件目は、**setosa_prob** が **0.91～** なので **setosa** に判定されています）。

また、**WITH** 句では、目的変数で指定した **Species** に「**_Pred**」を付けた列名も指定する必要があり、ここに判定結果（予測した種類）が出力されます。

WITH 句は、正確に列名を定義しないと、ステートメントの実行エラーとなってしまいます。どういった列名を指定しないといけないかは、次のように適当な列名を入れて実行することで確認することができるので、エラーが返された場合は、これを試してみてください。

R の場合のモデル保存（rxSerializeModel）

R を利用してモデルを保存する場合は、Python での **rx_serialize_model** 関数は、**rxSerializeModel** という名前、Python での「**realtime_scoring_only=True**」は「**realtimeScoringOnly=TRUE**」と指定するようにします（R では **TRUE** を全て大文字で記述することに注意してください）。

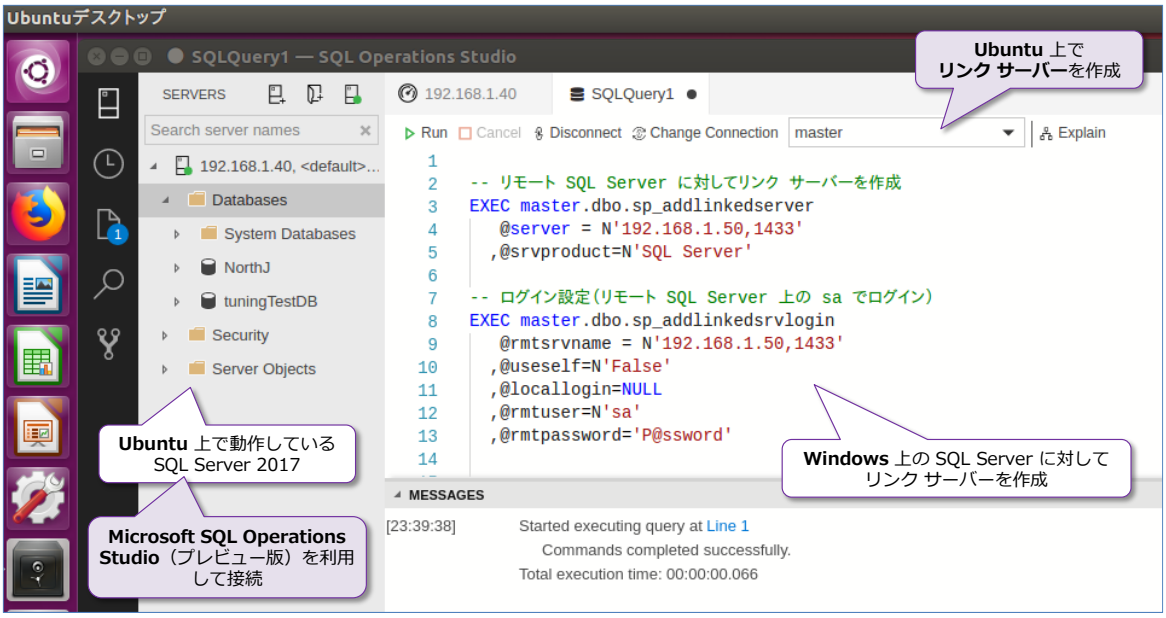
```
-- モデルをテーブルに INSERT
DECLARE @output_model varbinary(max)

EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
model1 <- rxDForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                    ,data = InputDataSet, nTree = 10)
output_model <- rxSerializeModel(model1, realtimeScoringOnly = TRUE)
'
,@input_data_1 = N'SELECT * FROM iris'
,@params = N'@output_model varbinary(max) OUTPUT'
,@output_model = @output_model OUTPUT

INSERT INTO t_model VALUES(@output_model, 'rxDForest R')

-- 格納されたモデルの確認
SELECT * FROM t_model
```

このように格納したモデルは、R の場合でも前掲の **PREDICT** 関数を利用して、Python のときと全く同じようにネイティブ スコアリングを実行することができます。

SQL Server 2017 on Linux で PREDICT 関数でネイティブ スコアリング

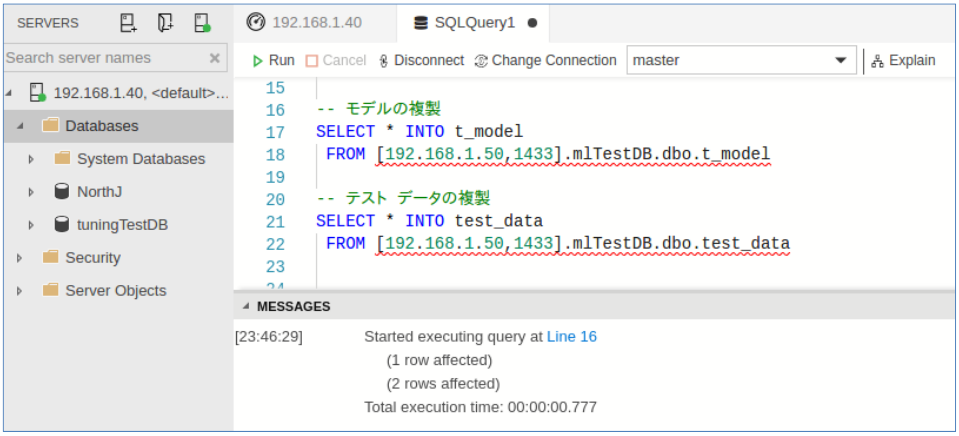
SQL Server 2017 on Linux では、Machine Learning Services はサポートされていませんが、**PREDICT** 関数はサポートされています。したがって、Windows 上の SQL Server 2017 で作成したモデルを、SQL Server 2017 on Linux にコピーすれば、ネイティブ スコアリングが可能です。SQL Server 2017 同士でのテーブルのコピー（モデルの複製）には、**リンク サーバー**機能を利用すると便利です。例えば、次のように **sp_addlinkedserver** と **sp_addlinkedsrvlogin** を利用することで、リンク サーバーを作成することができます（Windows 上の SQL Server の IP アドレスが **192.168.1.50** で、既定のインスタンスを利用している場合の作成例）。

```
-- リモート SQL Server に対してリンク サーバーを作成
EXEC master.dbo.sp_addlinkedserver
   @server = N'192.168.1.50,1433'
  ,@srvproduct=N'SQL Server'

-- ログイン設定（リモート SQL Server 上の sa でログイン）
EXEC master.dbo.sp_addlinkedsrvlogin
   @rmtsrvname = N'192.168.1.50,1433'
  ,@useself=N'False'
  ,@locallogin=NULL
  ,@rmtuser=N'sa'
  ,@rmtpassword='sa のパスワード'
```



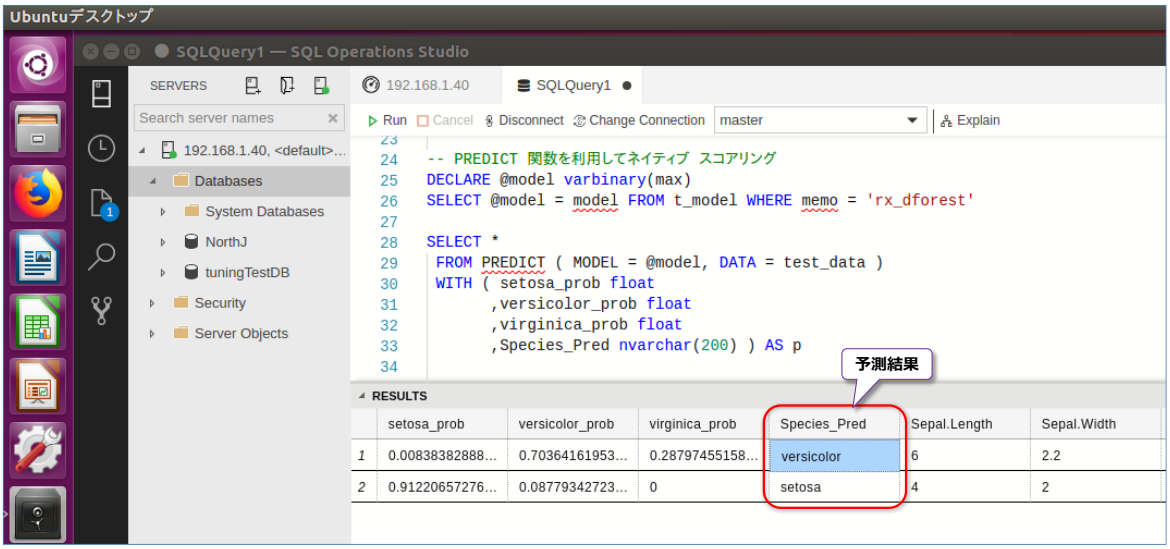
リンク サーバーを作成した後は、次のようにモデルとテスト データを丸ごとコピーします。

```
-- モデルの複製
SELECT * INTO t_model
 FROM [192.168.1.50,1433].mlTestDB.dbo.t_model

-- テスト データの複製
SELECT * INTO test_data
 FROM [192.168.1.50,1433].mlTestDB.dbo.test_data
```



モデルとテスト データの複製が完了したら、PREDICT 関数を利用して、ネイティブ スコアリングを実行してみます。

```
-- PREDICT 関数を利用してネイティブ スコアリング
DECLARE @model varbinary(max)
SELECT @model = model FROM t_model WHERE memo = 'rx_dforest'

SELECT *
 FROM PREDICT ( MODEL = @model, DATA = test_data )
 WITH ( setosa_prob float
       ,versicolor_prob float
       ,virginica_prob float
       ,Species_Pred nvarchar(200) ) AS p
```



Windows 上で PREDICT した結果と同じ結果を取得できたことを確認できると思います。このように、モデルをテーブルに保存しておけば、SQL Server 2017 on Linux を PREDICT 用（スコアリング用）のサーバーとして利用することができます。なお、Linux 上の SQL Server 2017 については、本自習書シリーズの「**No. 2 SQL Server 2017 on Linux**」編で詳しく説明しているので、こちらもぜひご覧いただければと思います。

また、ネイティブ スコアリングの性能については次の章で説明します。

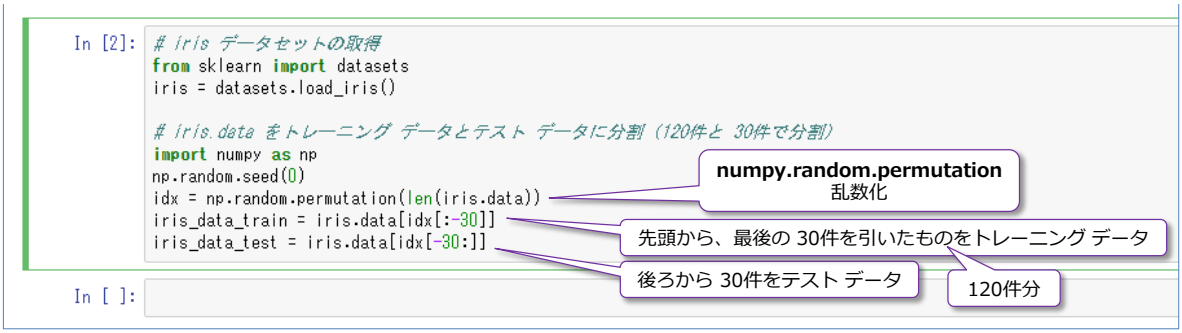
2.9 トレーニング データとテスト データの分割

これまでの手順では、**iris** データの全件（150 件すべて）をトレーニング データ（訓練データ）として、決定木やランダム フォレストのモデルを作成してきました。しかし、実際の機械学習では、既存のデータに対して、すべてをトレーニング データにするのではなく、既存のデータをトレーニング データとテスト データに分割して、モデルの精度を検証していくのが定番です。

例えば、Python 言語でよく利用されているのは、次のように **scikit-learn**（**sklearn**）ライブラリの **train_test_split** 関数でトレーニング データとテスト データを分割するという方法です。

この関数では、第 2 引数に「**test_size=0.2**」と指定すれば、テスト データを 20%、残りの 80%をトレーニング データに分割することができます。

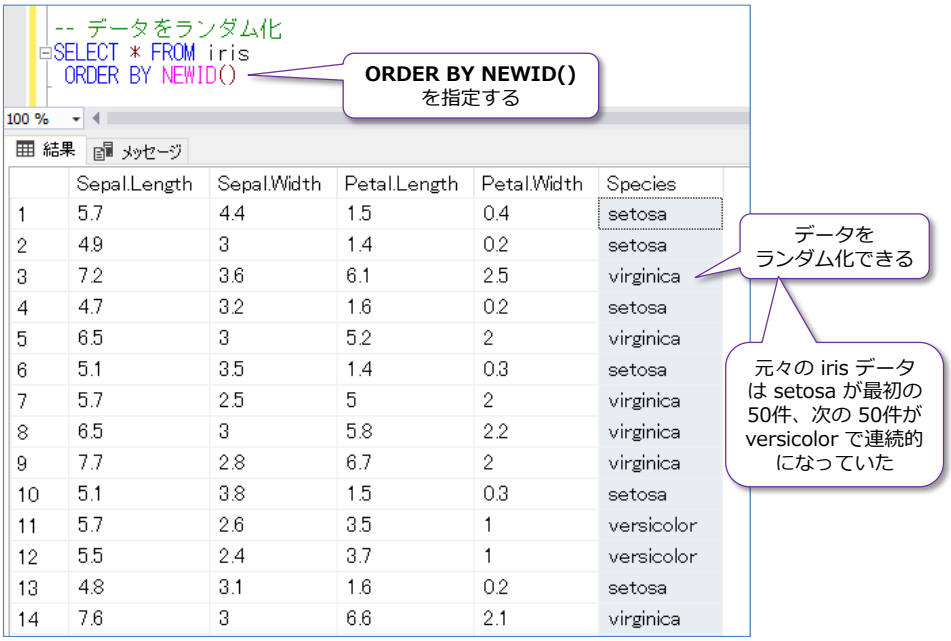
また、Python では、次のように **numpy** ライブラリの **random.permutation** 関数を利用して、データを乱数化して、トレーニング データとテスト データに分割するといった方法もよく利用されています。



SQL Server 上のデータをトレーニング データとテスト データに分割

トレーニング データとテスト データの分割には、データを乱数化して、ランダムに抽出することが重要になりますが、SQL Server のデータをランダムに抽出するには、**NEWID** 関数を利用すると簡単に行えます。これは、次のように利用できます。

```
-- データをランダム化
USE mlTestDB
SELECT * FROM iris
 ORDER BY NEWID()
```

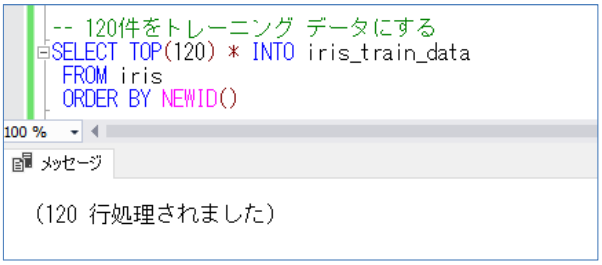


	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.7	4.4	1.5	0.4	setosa
2	4.9	3	1.4	0.2	setosa
3	7.2	3.6	6.1	2.5	virginica
4	4.7	3.2	1.6	0.2	setosa
5	6.5	3	5.2	2	virginica
6	5.1	3.5	1.4	0.3	setosa
7	5.7	2.5	5	2	virginica
8	6.5	3	5.8	2.2	virginica
9	7.7	2.8	6.7	2	virginica
10	5.1	3.8	1.5	0.3	setosa
11	5.7	2.6	3.5	1	versicolor
12	5.5	2.4	3.7	1	versicolor
13	4.8	3.1	1.6	0.2	setosa
14	7.6	3	6.6	2.1	virginica

NEWID 関数は、新しい **GUID** 値（グローバル一意識別子）を生成することができる関数なので、これを **ORDER BY** に指定することで、データをランダムに出力できるようになります。

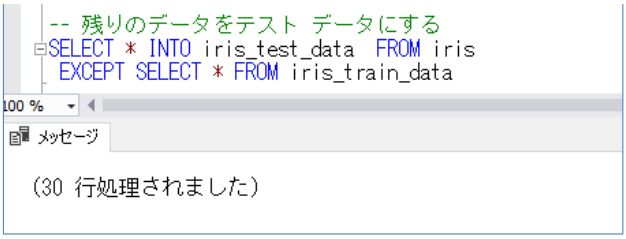
したがって、次のように **TOP(120)** を付けて、**SELECT .. INTO** で新しいテーブルを作成すれば、120 件のトレーニング データを作成することができます。

```
-- 120件をトレーニング データにする
SELECT TOP(120) * INTO iris_train_data
 FROM iris
 ORDER BY NEWID()
```



テスト データは、このトレーニング データを取り除いたものにすれば良いので、次のように **EXCEPT** を利用することで、残りのデータをテスト データにすることができます。

```
-- 残りのデータをテスト データにする
SELECT * INTO iris_test_data  FROM iris
 EXCEPT SELECT * FROM iris_train_data
```



このように、SQL Server を利用する場合でも、簡単にトレーニング データとテスト データを分割することができます。

なお、上の例では **120** という部分が固定値でしたが、比率に応じて、データを分割したい場合は、次のように Transact-SQL の変数を利用することで、簡単に行えます（80% をトレーニング データにする場合の例）。

```
-- トレーニング データの割合を指定（80% にする場合）
DECLARE @train_ratio int = 80
DECLARE @data_count int, @train_data_count int

SELECT @data_count = COUNT(*) FROM iris
SET @train_data_count = FLOOR(@data_count * (@train_ratio / 100.0))

-- トレーニング データ（80%）
SELECT TOP(@train_data_count) * INTO iris_train_data
 FROM iris
 ORDER BY NEWID()

-- テスト データ（20%）
SELECT * INTO iris_test_data
 FROM iris
  EXCEPT SELECT * FROM iris_train_data
```

トレーニング データとテスト データの分割が完了したら、後は、モデルを作成するときに、次のように **@input_data_1** パラメーターにトレーニング データ（**iris_train_data** テーブル）を指定するように変更します。

```
-- モデルの作成にトレーニング データを利用（スクリプトは前項と同じもの）
DECLARE @output_model varbinary(max)
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from revoscalepy import rx_dforest, rx_serialize_model
InputDataSet["Species"] = InputDataSet["Species"].astype("category")
model1 = rx_dforest("Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width"
                    ,data = InputDataSet, n_tree = 10)
output_model = rx_serialize_model(model1, realtime_scoring_only = True)
'
,@input_data_1 = N'SELECT * FROM iris_train_data'
,@params = N'@output_model varbinary(max) OUTPUT'
,@output_model = @output_model OUTPUT

INSERT INTO t_model VALUES(@output_model, 'rx_dforest train_data')
```

```
-- モデルの作成にトレーニング データを利用
DECLARE @output_model varbinary(max)
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from revoscalepy import rx_dforest, rx_serialize_model
InputDataSet["Species"] = InputDataSet["Species"].astype("category")
model1 = rx_dforest("Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width"
                    ,data = InputDataSet, n_tree = 10)
output_model = rx_serialize_model(model1, realtime_scoring_only = True)
'
,@input_data_1 = N'SELECT * FROM iris_train_data'
,@params = N'@output_model varbinary(max) OUTPUT'
,@output_model = @output_model OUTPUT

INSERT INTO t_model VALUES(@output_model, 'rx_dforest train_data')

-- 確認
SELECT * FROM t_model
```

トレーニング データ
を指定してモデル作成

100 %

結果 メッセージ

	model	memo
1	0x626C6F62AF27C50D5C06A6A8447C7499DB028388FAD35A15621B...	rx_dforest
2	0x626C6F628C4B4C8D752A576B91DE52A72AA9E94936C3FB98E561...	rxDForest R
3	0x626C6F622CC8B31566BC249EA98306B1DEC1DFFF369B576F9E9...	rx_dforest train_data

あとは、テスト データを利用して、PREDICT（予測）を実行すれば、今度は正解データがあるので、モデルの精度を検証するために利用することができます。

```
-- テスト データでモデルの精度を検証
DECLARE @model varbinary(max)
SELECT @model = model FROM t_model WHERE memo = 'rx_dforest train_data'

SELECT *
 FROM PREDICT ( MODEL = @model, DATA = iris_test_data )
 WITH ( setosa_prob float
       ,versicolor_prob float
       ,virginica_prob float
       ,Species_Pred nvarchar(200) ) AS p
```

テスト データ
を指定して予測を実行

予測結果 **精度の検証に利用** **正解データ**

100 %

結果 メッセージ

	setosa_prob	versicolor_prob	virginica_prob	Species_Pred	Sepal...	Sepal....	Petal....	Petal...	Species
7	1	0	0	setosa	5	3.6	1.4	0.2	setosa
8	1	0	0	setosa	5.1	3.8	1.9	0.4	setosa
9	1	0	0	setosa	5.3	3.7	1.5	0.2	setosa
10	1	0	0	setosa	5.4	3.4	1.5	0.4	setosa
11	1	0	0	setosa	5.4	3.9	1.7	0.4	setosa
12	0.8095463...	0.1822869060...	0.00816676...	setosa	5.5	3.5	1.3	0.2	setosa
13	0.0095463...	0.9298040786...	0.06064958...	versicolor	5.6	2.7	4.2	1.3	versicolor
14	0.0095463...	0.9298040786...	0.06064958...	versicolor	5.7	2.8	4.1	1.3	versicolor
15	0.0095463...	0.9298040786...	0.06064958...	versicolor	5.8	2.7	3.9	1.2	versicolor
16	0.8095463...	0.1822869060...	0.00816676...	setosa	5.8	4	1.2	0.2	setosa
17	0	0.1178475559...	0.88215244...	virginica	5.9	3	5.1	1.8	virginica
18	0.0045777...	0.5564910965...	0.43893116...	versicolor	6	3	4.8	1.8	virginica
19	0.0095463...	0.9298040786...	0.06064958...	versicolor	6	3.4	4.5	1.6	versicolor
20	0.0095463...	0.9460298281...	0.04442383...	versicolor	6.1	2.8	4.7	1.3	versicolor

不正解

STEP 3. Python を利用した機械学習

この STEP では、SQL Server 2017 に統合された Python を利用して機械学習を行う手順をもう少し詳しく説明します。通常の Python との違いや、scikit-learn を利用したいろいろな機械学習アルゴリズムの利用方法、pickle によるモデルの保存、Microsoft Cognitive Toolkit を利用した画像認識などを説明します。

この STEP では、次のことを学習します。

- ✓ sp_execute_external_script の復習
- ✓ バージョン確認、利用可能なライブラリの確認（conda.exe）
- ✓ 追加のライブラリのインストール（pip.exe）
- ✓ scikit-learn（sklearn）で機械学習
- ✓ pickle によるモデルの保存、予測結果の保存
- ✓ モデルの保存方法の違い（pickle vs. rx_serialize_model）
- ✓ Microsoft Cognitive Toolkit（CNTK）を利用した画像認識

3.1 SQL Server 2017 に統合された Python

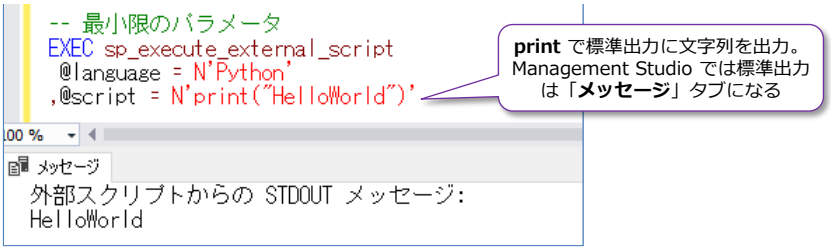
前の章では、**Revoscalepy**（RevoScaleR の Python 版）を利用して、**rx_dforest** によるランダム フォレストのモデルの作成や **rxPredict** による予測の実行、**rx_serialize_model** によるモデルの保存（シリアライズ化）、**PREDICT** 関数によるネイティブ スコアリングなどを説明しました。

sp_execute_external_script の復習

改めて、**sp_execute_external_script** システム ストアド プロシージャの利用方法を確認すると、次のようになります。

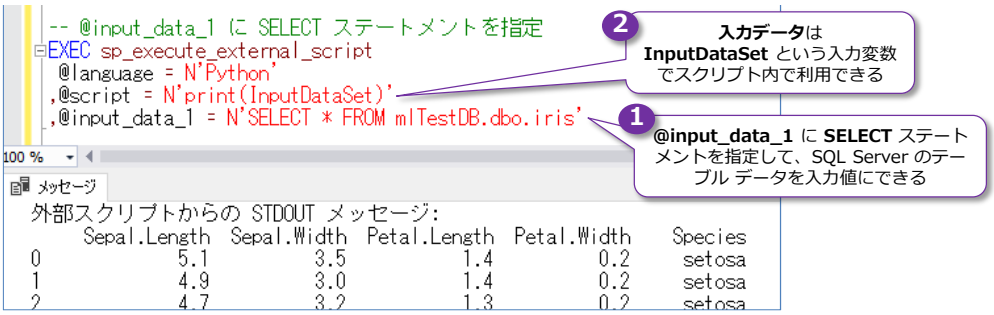
```
EXEC sp_execute_external_script
       @language = N'Python'
     , @script = N'実行したい Python スクリプト'
     , @input_data_1 = N'Python で処理したい入力データ'
     , @input_data_1_name = N'input_data_1 に対して設定する名前。既定値は InputDataSet'
     , @output_data_1_name = N'Python スクリプトで処理した結果。既定値は OutputDataSet'
     , @params = N'追加の変数定義。結果を変数で受け取る場合などに利用'
     WITH RESULT SETS (( 列名 データ型 null/not null, ・・・ ))
```

Python スクリプトを実行するには、**@language** で「**Python**」を指定して、**@script** に任意のスクリプトを記述します。この 2 つが必須パラメーターで、次のように実行可能です。

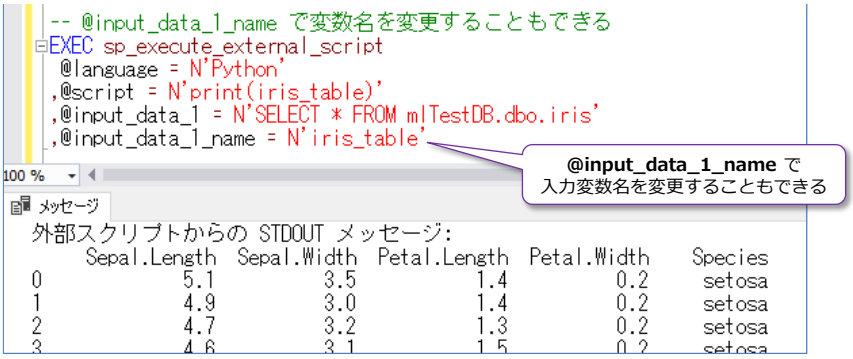


なお、**N''** の **N** は、SQL Server における Unicode 文字列を扱うためのプレフィックス（National の N）で、各パラメーターに与える文字列は **N** で囲んでおく必要があります。

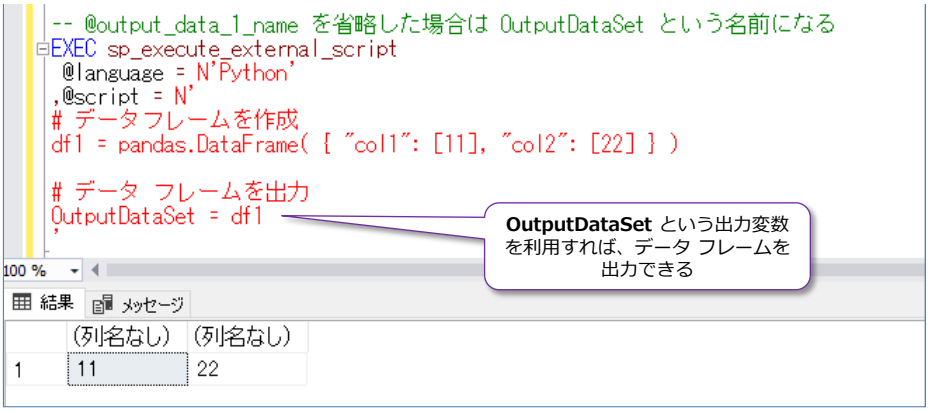
@input_data_1 では、**SELECT** ステートメントなどを記述して、SQL Server 上のテーブル データをスクリプトに与えることができ、スクリプト内では **InputDataSet** という名前（入力変数）で利用することができます。

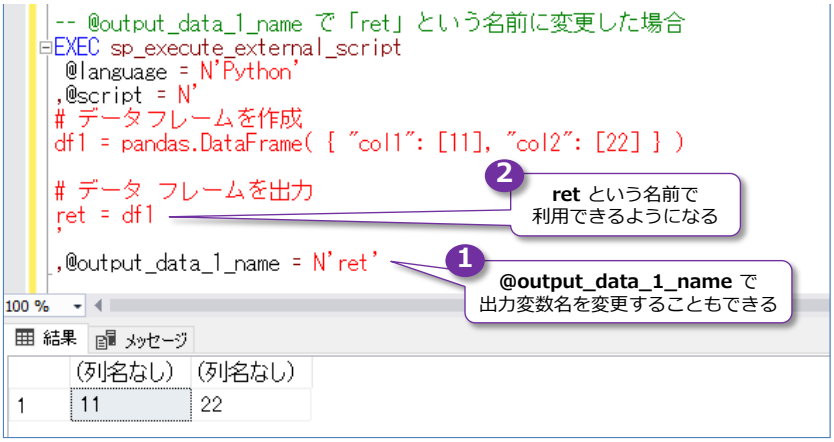


InputDataSet という名前を変更したい場合には、**@input_data_1_name** で変数名を指定することもできます。



@output_data_1_name は、出力変数の名前で、省略した場合は、**OutputDataSet** という名前が補われます。





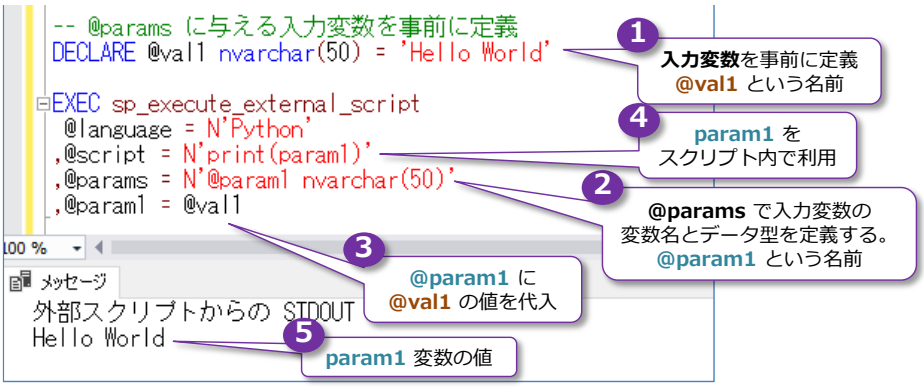
@params での入力変数／出力変数

sp_execute_external_script の **@params** では、スクリプト内で利用できる追加の変数（入力変数または出力変数）を定義することができます。前章ではネイティブ スコアリングのためのモデルを出力するために利用しました。「**@変数名 データ型 OUTPUT**」と変数を定義することで、出力変数にすることができ、スクリプト内では @ なしの「**変数名**」で利用することができます（モデルの出力では **@output_model** という名前の変数を利用しました）。

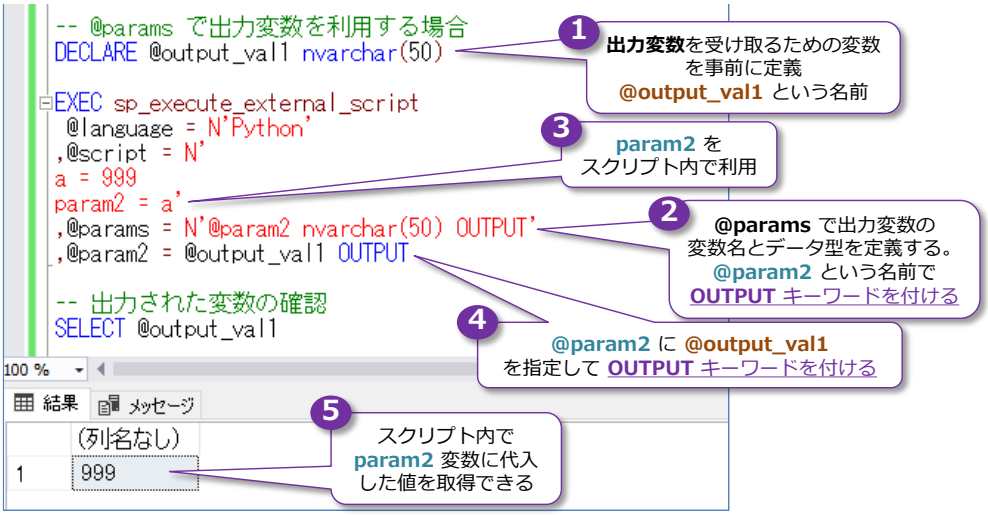
@params で定義した変数には、別途、どういった値を代入するのか、あるいは出力させたいのかを次のように追加定義する必要があります。

```
    , @params = N'@変数１ データ型 OUTPUT, @変数２ データ型, …'
    , @変数１ = @出力用の変数名 OUTPUT
    , @変数２ = @入力用の変数名
    ,   :
```

入力用の変数を利用する場合は、次のように事前に定義しておくようにします。



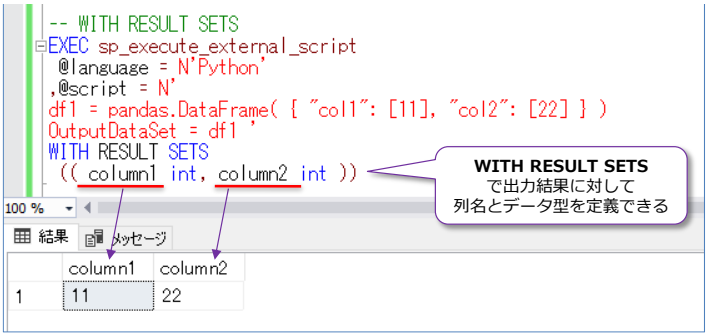
出力用の変数を利用する場合は、次のように **OUTPUT** キーワードを利用するようにします。



前章では、上のように出力変数を利用して、モデル（**model1**）を **varbinary(max)** データ型で定義して出力していました（出力したものを **INSERT** ステートメントでテーブルに格納）。

WITH RESULT SETS

sp_execute_external_script の **WITH RESULT SETS** は、**OutputDataSet** または **@output_data_1_name** で指定した出力変数（データ フレーム形式）の出力結果に対して、次のように列名やデータ型を指定することができます。



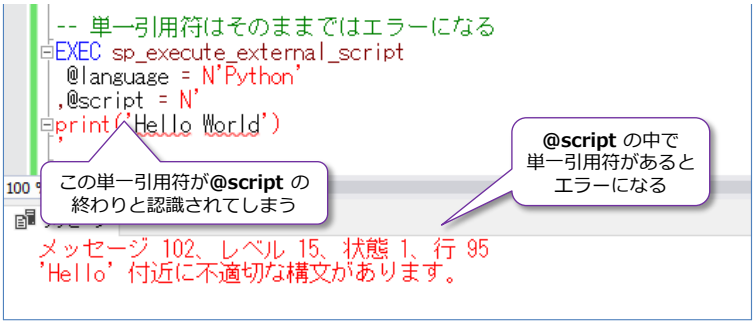
なお、**WITH RESULT SETS** では、「**WITH RESULT SETS (AS OBJECT テーブル名)**」という形で、既存のテーブル スキーマ（列名やデータ型）を利用して、結果を出力することもできます。

通常の Python スクリプトとの違い

sp_execute_external_script で指定する Python スクリプトと、通常の Python スクリプトの違いは、ほとんどありませんが、主に次のことが考慮事項になります。

- **Python** のバージョンは **3.5.2**（**Anaconda** ディストリビューションを利用）
- 単一引用符は、単一引用符を **2 つ**に変更するか、**二重引用符**に変更する
- pandas ライブラリは、「**import pandas**」を付けるとエラーになる。
  pandas という名前でそのまま利用できる。あるいは「**import pandas as pd**」のように別名を定義して利用するようにする
- 追加のライブラリは、通常の Python と同様、**pip** でインストールできる。
  pip の場所は、SQL Server 配下のフォルダーになる（後述）
- **Anaconda** がインストールされているので、**pip** でインストールしなくても利用できるライブラリがある（numpy や scikit-learn、pickle、scipy などは import だけで利用できる。利用できるライブラリの一覧については後述）。

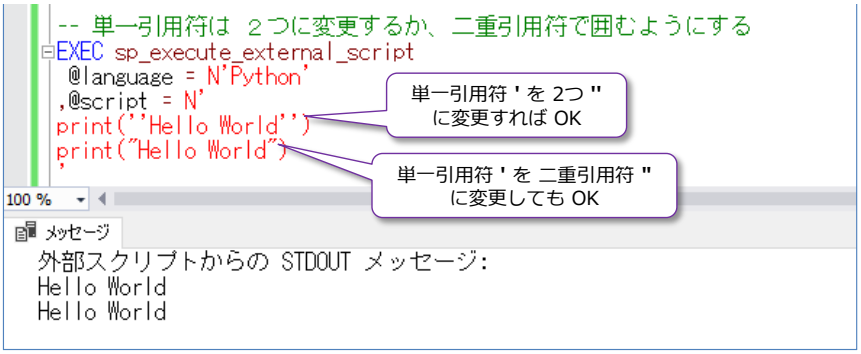
単一引用符については、次のような状況です。



sp_execute_external_script では、**@script** で「**N' ～ '**」という形でスクリプトを記述するので、スクリプト内に単一引用符があると、そこで **@script** の引数が終わりと見なされてしまって、ストアド プロシージャの実行エラーになってしまいます。

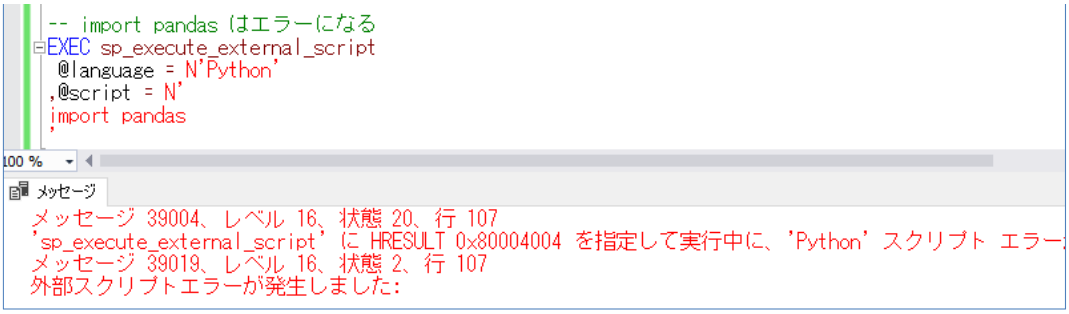
Python では、文字列を囲む際に**単一引用符**または**二重引用符**を利用しますが、単一引用符を利用

しているユーザーが多いと思います。そうしたスクリプトを **@script** に貼り付けて実行しようとすると、このようなエラーになってしまいます。これを回避するには、スクリプト内の単一引用符を 2 つに変更するか、二重引用符に変更するようにします。



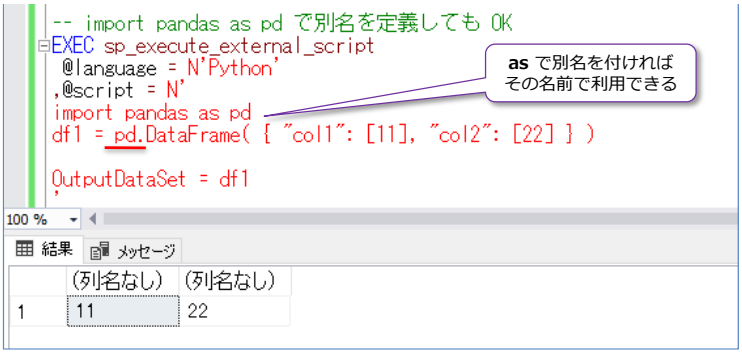
2 章の Python スクリプトでは、二重引用符を利用していましたが、二重引用符であれば、通常の Python スクリプトでも動作させることができるので、通常でも、**sp_execute_external_script** でも、両方で動作が可能な二重引用符で囲むのがお勧めの方法になります。通常の Python の開発環境である **Jupyter Notebook** や **PyCharm**、**Visual Studio Code** などで、二重引用符でスクリプトを記述しておけば、**sp_execute_external_script** への移植が容易になります。

pandas ライブラリに関しては、次のように「**import pandas**」を付けるとエラーになります。



2 章で利用したように「**pandas.DataFrame**」という形で、「**import pandas**」を利用しなくても、**pandas** という名前でそのまま利用することができます。

あるいは、次のように「**import pandas as pd**」という形で別名を定義しても利用することができます。



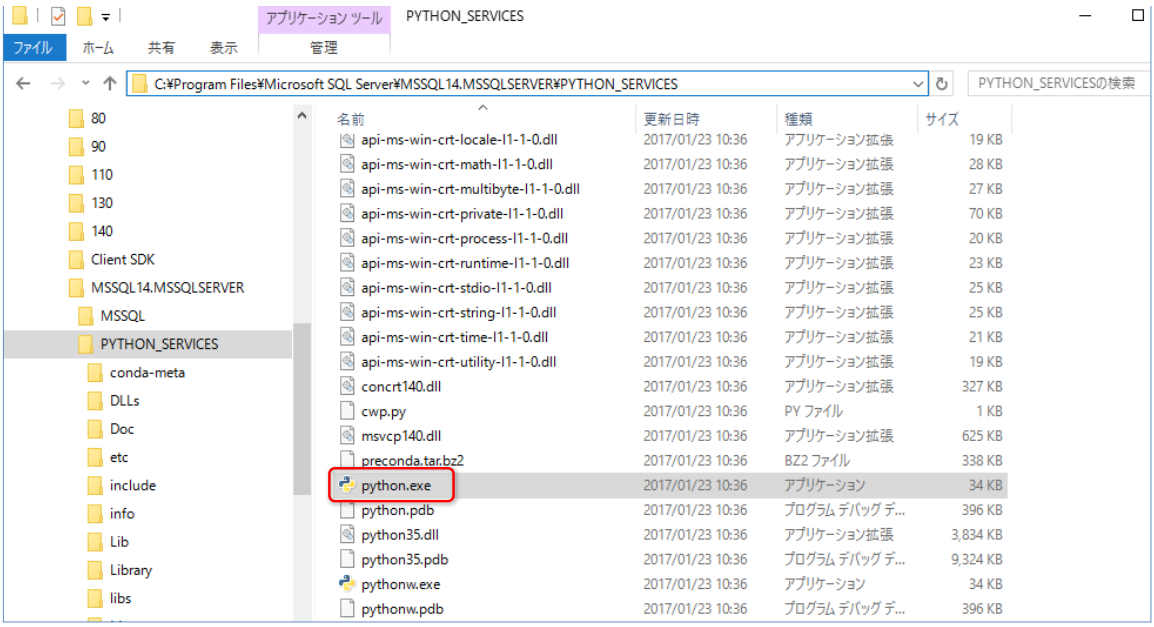
3.2 バージョン確認、利用可能な Python ライブラリの一覧

SQL Server 2017 に統合された Python は、Python のディストリビューションの 1 つである「**Anaconda**」が利用されています。

Python のバージョンの確認（Python 3.5.2）

Python インタープリターである「**python.exe**」は、以下のフォルダーに格納されています（既定のインスタンスとして SQL Server 2017 をインストールした場合）。

C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES



SQL Server 2017 を名前付きインスタンスとしてインストールしている場合は、「**MSSQL14.MSSQLSERVER**」の部分が「**MSSQL14.インスタンス名**」に変わります。

Python のバージョンを確認するには、コマンド プロンプトから次のように実行します。

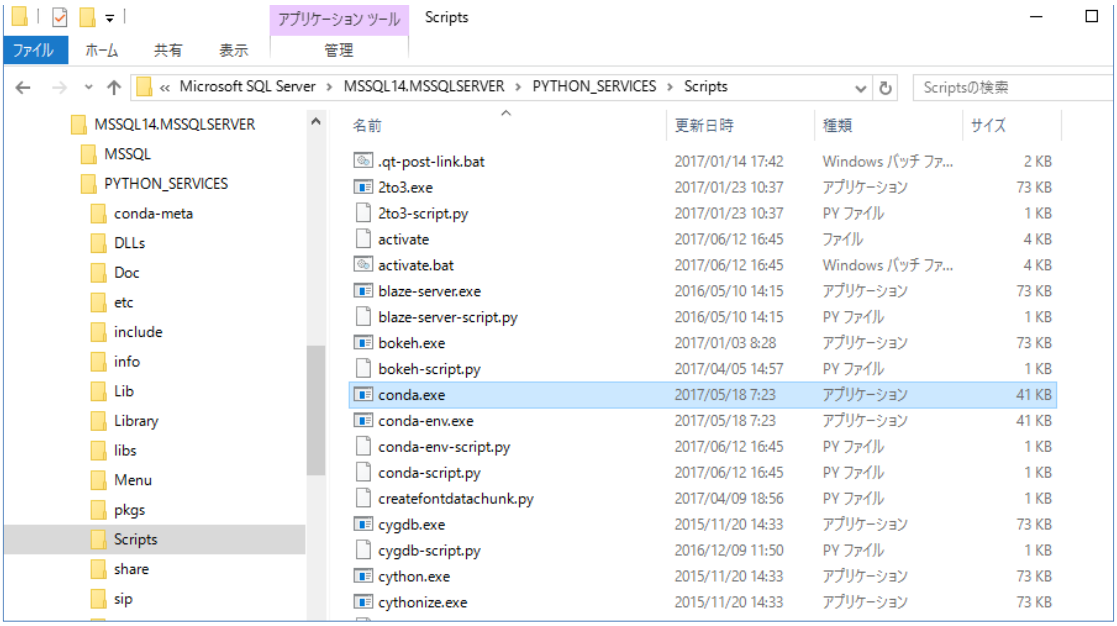
```
cd C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES
python -V
```

-V の **V** は大文字で指定します。結果は「**Python 3.5.2**」と表示されて、Python のバージョンが「**3.5.2**」であることを確認できます。

Anaconda のバージョンの確認（4.3.22）

Anaconda のコマンド群（**conda.exe** や **pip.exe** など）は、以下のフォルダー（**PYTHON_SERVICES** フォルダーの下の **Scripts** フォルダー）に格納されています（既定のインスタンスとして SQL Server 2017 をインストールした場合）。

C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES¥Scripts



Anaconda のバージョンは、「**conda -V**」（V は大文字）と実行することで確認することができます（コマンド プロンプトから次のように実行します）。

```
cd C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES¥Scripts
conda -V
```

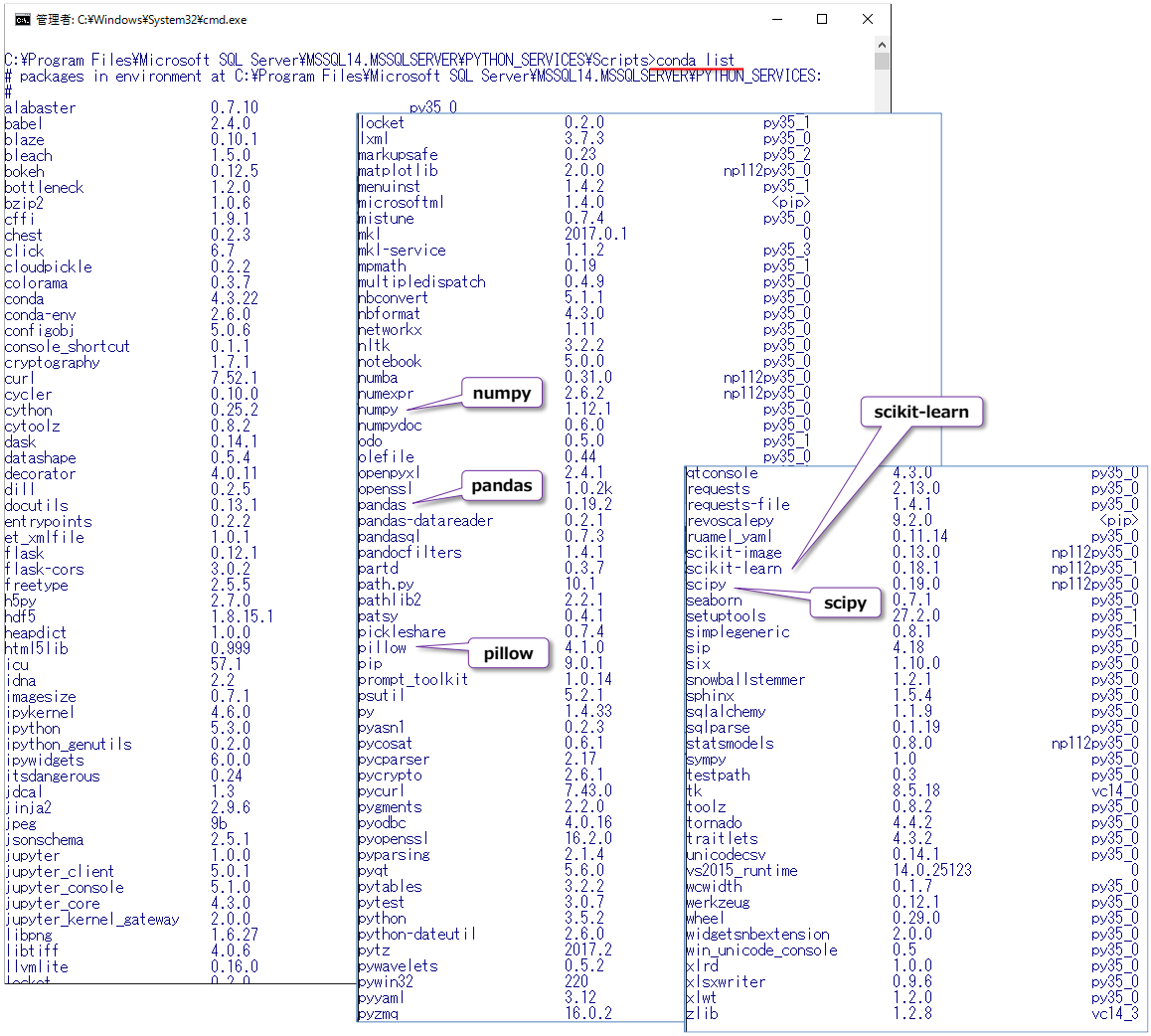
結果は「**conda 4.3.22**」と表示されて、Anaconda のバージョンが「**4.3.22**」であることを確認できます。

利用可能なライブラリの一覧

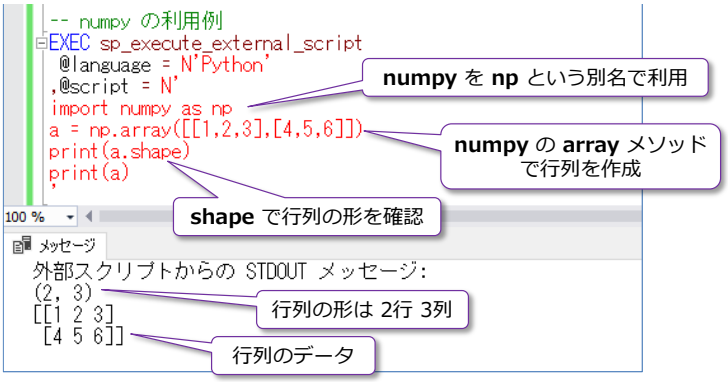
Anaconda は、Python でよく利用されるライブラリがインストール済みのディストリビューションになっているので、**numpy** や **scikit-learn**、**pickle**、**scipy**、**PIL**（pillow）といったメジャーなものは、別途インストールする必要なく利用することができます。こうした利用可能なライブ

ラリの一覧は、次のように「**conda list**」と実行することで確認することができます。

```
conda list
```



この一覧にリストされているものは、既にインストール済みなので、「**import ライブラリ名**」という記述で利用することができます。例えば、**numpy** を利用する場合は、次のように記述できます。



インストール済みではないライブラリであっても、後述の「**pip.exe**」を利用して、別途インストールを行えば、**import** で利用できるようになります。

3.3 追加のライブラリのインストール（pip install ～）

ディープ ラーニング（深層学習）での**定番フレームワーク**である「**Microsoft Cognitive Toolkit**（**CNTK**）」や「**Chainer**」、「**Google TensorFlow**」、「**Theano**」などは、通常の Python と同様、「**pip.exe**」を利用して別途インストールを行うことで利用できるようになります。

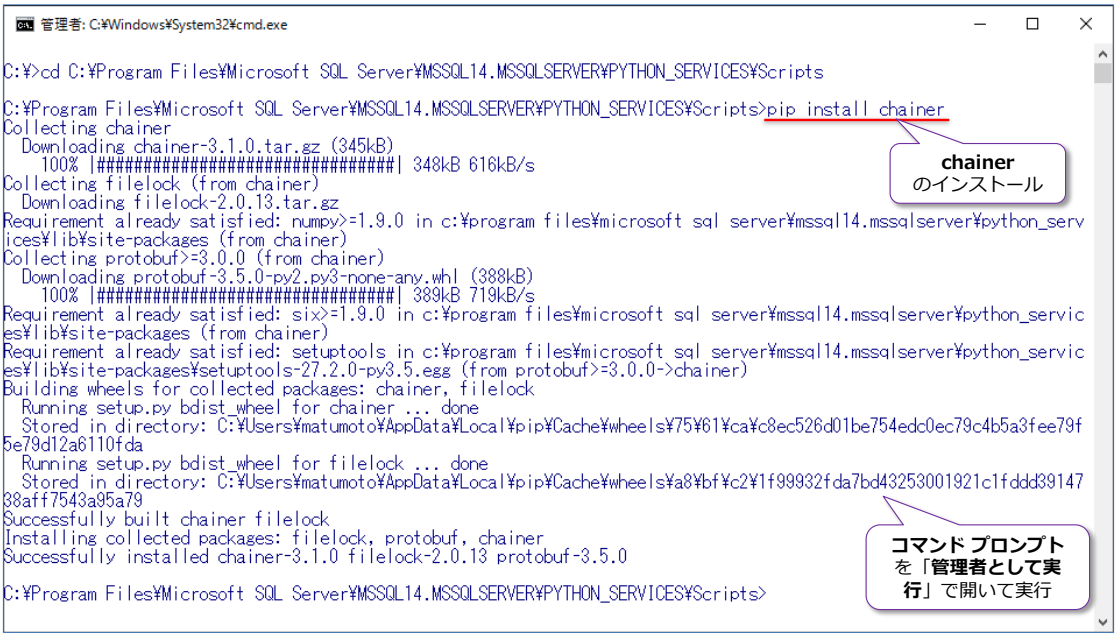
pip.exe の場所は、前掲の **conda.exe** の場所と同様、以下のフォルダーになります（既定のインスタンスとして SQL Server 2017 をインストールした場合）。

C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES¥Scripts

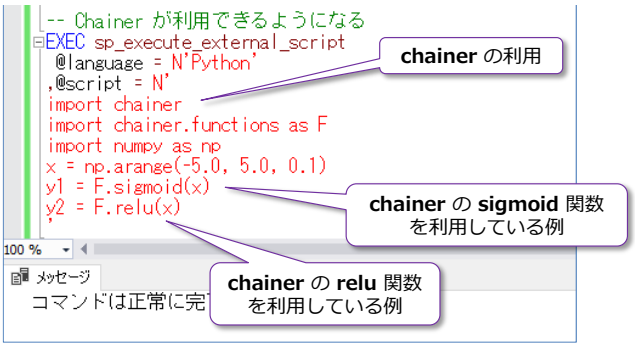
pip.exe は、基本的には「**pip install ライブラリ名**」という形で追加のライブラリをインストールすることができますが、詳しいインストール方法は、各ライブラリが提供さいている Web サイトで確認しておく必要があります。

Chainer の場合は、次のようにインストールすることができます。

```
cd C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES¥Scripts
pip install chainer
```



インストール後は、次のように **Chainer** を利用できるようになります。



3.4 scikit-learn（sklearn）で機械学習

2 章では、**Revoscalepy** の **rx_dforest** を利用したランダム フォレストの説明をしましたが、ここでは、Python での機械学習でよく利用される **scikit-learn**（**sklearn**）を利用して、ランダム フォレストやサポート ベクター マシン（SVM）、確率的勾配降下法（SGD）、ニューラル ネットワークなどを利用して、モデルの作成および予測を実行する方法を説明します。

scikit-learn は、2 章では **iris** データの利用方法を説明しました（**datasets.load_iris** メソッドでデータを取得して、**iris.data** に Sepal や Petal、**iris.target** にアヤメの種類が数値化されたものが格納されていました）。

ランダム フォレスト（RandomForestClassifier）

まずは、**scikit-learn** でランダム フォレストのモデルを作成できる **RandomForestClassifier** クラスを利用してみます。Python スクリプトは、次のように記述します。

```
# iris データの取得
from sklearn import datasets
iris = datasets.load_iris()

# モデルの作成
from sklearn.ensemble import RandomForestClassifier
model1 = RandomForestClassifier(n_estimators = 10)
model1.fit(iris.data, iris.target)

# 予測 (predict)
data1 = [[ 6.0
          ,2.2
          ,5.0
          ,1.5 ]]
print(model1.predict(data1))
```

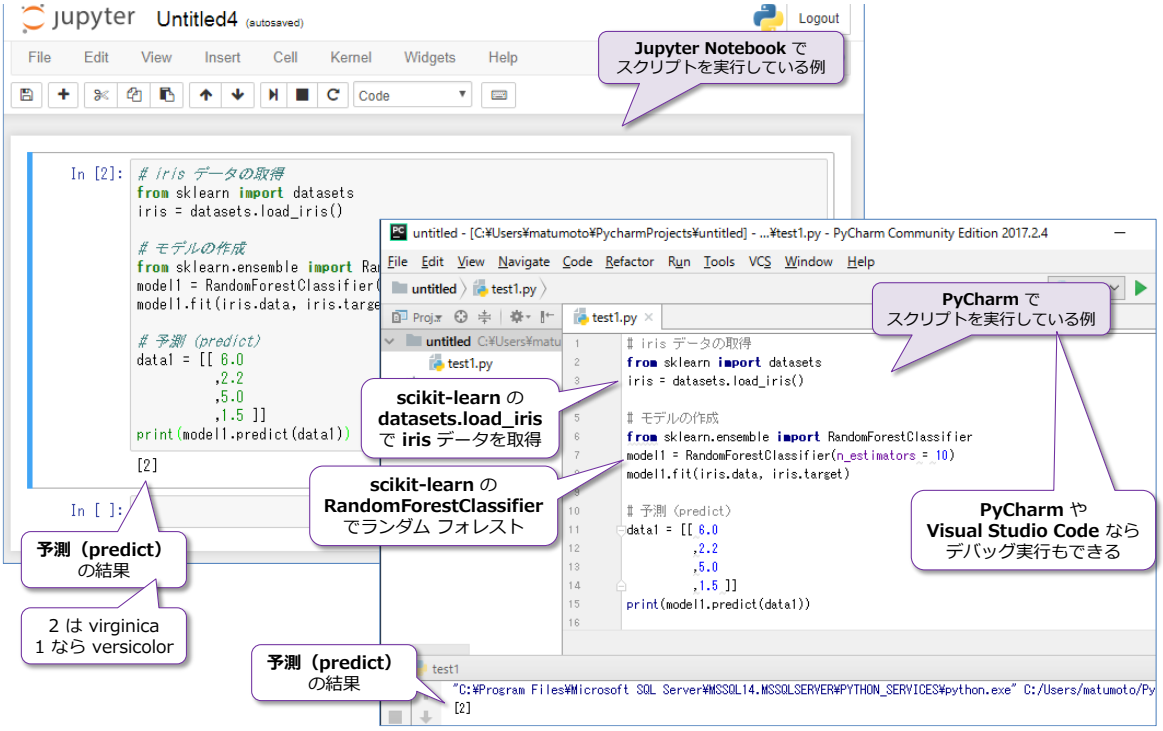
RandomForestClassifier クラスは、「**n_estimators = 10**」を指定することで、ツリーの数を指定できます（既定値は 10 なので、=10 の場合は省略することもできます）。このクラスは、**fit** メソッドでモデルを作成することができ、「**.fit(説明変数, 目的変数)**」のように利用します。Revoscalepy の **rx_dforest** では、「**rx_dforest("目的変数 ~ 説明変数", data=…)**」という形（チルダを利用）でモデルを作成しましたが、**fit** メソッドではチルダは不要で、第 1 引数に説明変数、第 2 引数に目的変数を指定します（カンマで区切って指定します）。

説明変数に指定した **iris.data** には、Sepal や Petal の Length／Width が格納されていて、目的変数に指定した **iris.target** には、アヤメの種類が数値化されたものが格納されています。これは 0 が setosa、1 が versicolor、2 が virginica になっているものです。

予測（**predict**）では、モデルを作成したときと同じデータ形式のものを与えますが、説明変数に指定した **iris.data** と同じデータ形式にします（**rx_dforest** では **Species** に **dummy** という文字列を入れた、データ フレーム形式のデータを作成しましたが、今回は Species は不要です）。

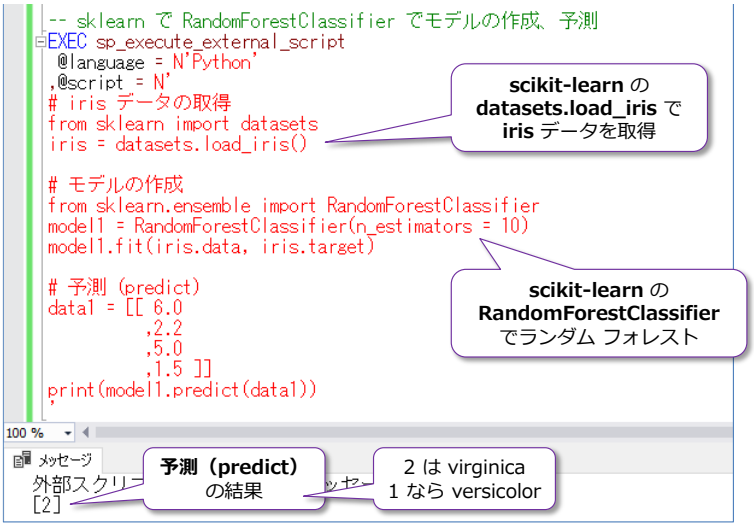
また、**iris.data** はデータ フレームではなく、**配列**（正確には numpy 配列）形式なので、予測に利用するデータ（**data1**）を「**[[6.0, 2.2, 5.0, 1.5]]**」という形で与えています。**predict** に関しては、2 章では「**predict(モデル名, データ)**」という形で利用してきましたが、**scikit-learn** では「**モデル名.predict(データ)**」という形で利用します。

以上の Python スクリプトは、通常の Python スクリプトと何も変わらないので、Python でよく利用される開発環境である **Jupyter Notebook** や **PyCharm**、**Visual Studio Code** などを利用して、実行およびデバッグを行うこともできます。



このように、SQL Server 2017 の Machine Learning Services の固有の機能を利用しない場合には、通常の Python 環境で開発／デバッグを行ってから、スクリプトを **sp_execute_external_script** の **@script** にコピーするといった使い方ができます。

sp_execute_external_script では、次のように実行します。

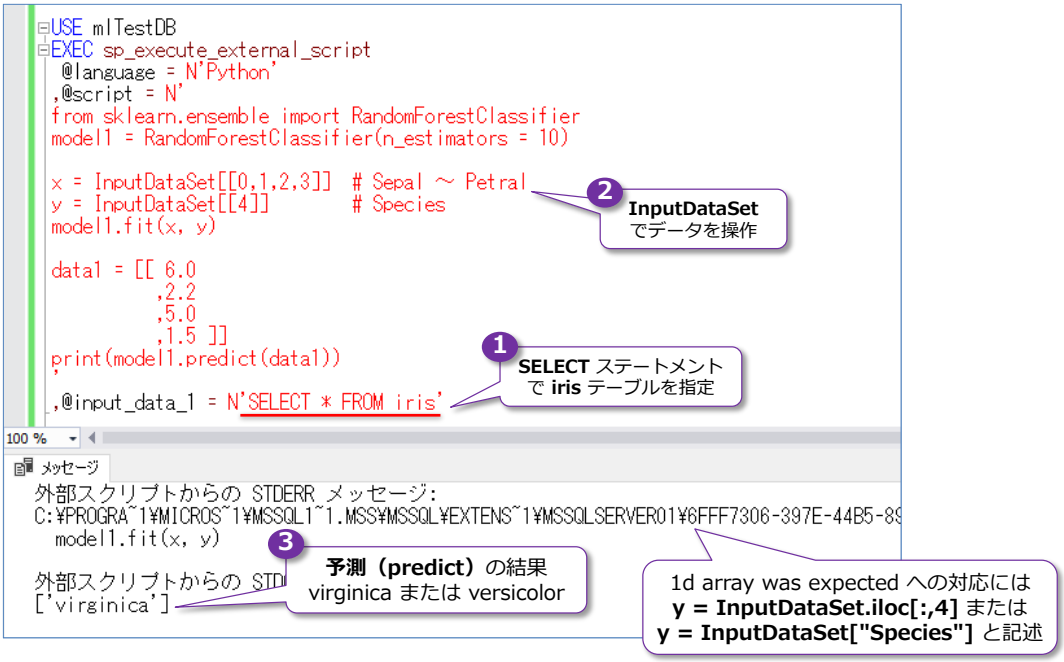


iris.target のアヤメの種類は、**0** が **setosa**、**1** が **versicolor**、**2** が **virginica** になっているので、結果が 2 の場合は virginica と判定されています。

@input_data_1 に SQL Server のデータを指定する場合

次に、**@input_data_1** に SQL Server のデータ（2 章で作成した **mlTestDB** データベース内の **iris** テーブル）を指定する場合のスクリプトに変更してみます。

```
USE mlTestDB
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from sklearn.ensemble import RandomForestClassifier
model1 = RandomForestClassifier(n_estimators = 10)

x = InputDataSet[[0,1,2,3]]  # Sepal ～ Petral
y = InputDataSet[[4]]        # Species
model1.fit(x, y)

data1 = [[ 6.0
          ,2.2
          ,5.0
          ,1.5 ]]
print(model1.predict(data1))
'
,@input_data_1 = N'SELECT * FROM iris'
```



前のスクリプトとの違いは、**InputDataSet**（入力変数）を利用しているところです。説明変数となる（Sepal や Petal）は、1 列目～4 列目のデータになるので「**InputDataSet[[0,1,2,3]]**」という形で **x** という変数に代入しています。InputDataSet は、pandas の DataFrame 形式にな

りますが、「**[[0,1,2,3]]**」と指定することで、1 列目～4 列目のデータを取得できます（Python では **0** から数えるので、0, 1, 2, 3 と指定しています）。数値ではなく、列名を指定することもできるので、「**[["Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width"]]**」と記述することもできます。

目的変数となる **Species** は 5 列目のデータになるので「**InputDataSet[[4]]**」と指定して、**y** という変数に代入しています。実は、この指定方法だと、「**1d array was expected**（1 次元の配列が期待されている）」という警告が返されるのですが、モデルの作成は問題なく行えます。この警告を出さないようにするには「**y=InputDataSet.iloc[:,4]**」や「**y=InputDataSet["Species"]**」と記述するようにします。**iloc** は pandas のデータ フレームを操作するためのメソッドで、**[行番号:列番号]** でデータを取得することができます。

モデルの作成は、「**model1.fit(x, y)**」という形で **x**（Sepal や Petal）と **y**（Species）を与えています。機械学習の世界では、この「**.fit(x, y)**」という使い方はよく利用されていて、説明変数に **x** や **X**、目的変数に **y** や **Y** という変数名を割り当てたりします。数学（数式）での「**y = x**」形式（y を求めるための x、アヤメの種類を求める／予測するための Sepal や Petal）です。

scikit-learn のサポート ベクター マシン（SVM）を利用する場合

次に、トレーニング データの件数が少ない場合の **classification**（分類）でよく利用される**サポート ベクター マシン**（SVM）という機械学習のアルゴリズムを利用してみます。詳しい利用方法は、以下の scikit-learn サイトのヘルプに記載されています。

1.4. Support Vector Machines
http://scikit-learn.org/stable/modules/svm.html

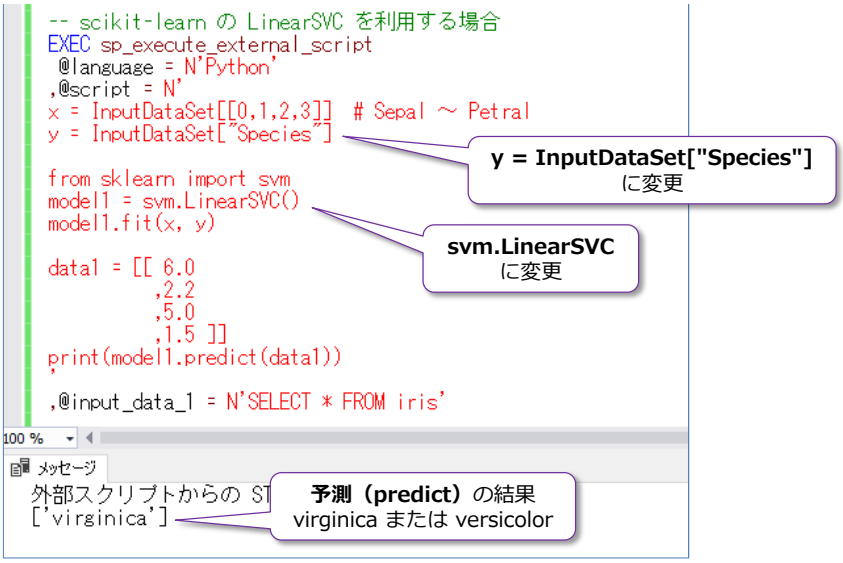
scikit-learn では **LinearSVC** というクラスを利用しますが、前掲の **RandomForestClassifier** クラスの場合と利用方法はほとんど同じで、次のように記述します（**y=** の部分は列名指定に変更しています）。

```
-- scikit-learn の LinearSVC を利用する場合
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
x = InputDataSet[[0,1,2,3]]  # Sepal ～ Petral
y = InputDataSet["Species"]

from sklearn import svm
model1 = svm.LinearSVC()
model1.fit(x, y)

data1 = [[ 6.0
          ,2.2
          ,5.0
          ,1.5 ]]
print(model1.predict(data1))
```

```
'
,@input_data_1 = N'SELECT * FROM iris'
```

変更箇所はたったの 2 行で、「**from sklearn import svm**」と「**model1 = svm.LinearSVC()**」に変更しているだけです。あとのコードはランダム フォレストのときと全く同じです。

サポート ベクター マシン（SVM）は、昨今のようにディープ ラーニングが流行する前の、画像認識のアルゴリズムとしてよく利用されていたものです。精度が高い反面、データの件数が増えた場合に計算量が増大してしまうので、性能面での課題があったりします。機械学習のアルゴリズムは、それぞれ一長一短があり、どのアルゴリズムが最適なのかは、データの特性や何を予測したいのかによっても変わってくるので、いろいろなアルゴリズムを実際のデータで検証してみることが重要になります。



scikit-learn の SGD（Stochastic Gradient Descent）での分類

次に、トレーニング データの件数が多い場合の **classification**（分類）で利用することが多い「**Stochastic Gradient Descent**」（SGD：確率的勾配降下法）の分類モデル（**SGDClassifier** クラス）を利用してみます。詳しい利用方法は、以下の scikit-learn サイトのヘルプに記載されています。

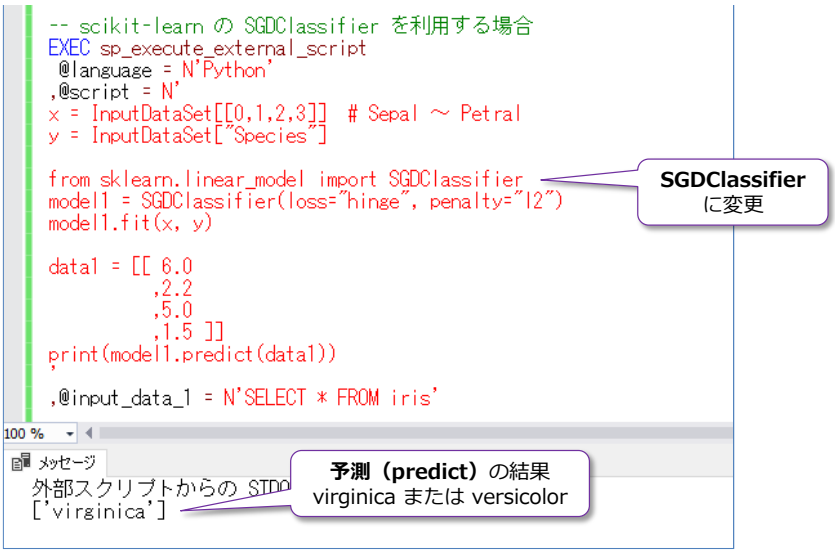
1.5. Stochastic Gradient Descent
http://scikit-learn.org/stable/modules/sgd.html

SGDClassifier クラスの利用方法は、**LinearSVC** クラスや **RandomForestClassifier** クラスの場合とほとんど同じです（次のように記述します）。

```
-- scikit-learn の SGDClassifier を利用する場合
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
x = InputDataSet[[0,1,2,3]]  # Sepal ～ Petral
```

```
y = InputDataSet["Species"]

from sklearn.linear_model import SGDClassifier
model1 = SGDClassifier(loss="hinge", penalty="l2")
model1.fit(x, y)

data1 = [[ 6.0
          ,2.2
          ,5.0
          ,1.5 ]]
print(model1.predict(data1))
'
,@input_data_1 = N'SELECT * FROM iris'
```

「**from sklearn.linear_model import SGDClassifier**」と「**model1 = SGDClassifier(~)**」のところを変更しているだけです。



scikit-learn のニューラル ネットワークを利用する場合（MLPClassifier）

次に、**ニューラル ネットワーク**を利用した分類が可能な scikit-learn の **MLPClassifier** クラスを利用してみます。詳しい利用方法は、以下の scikit-learn サイトのヘルプに記載されています。

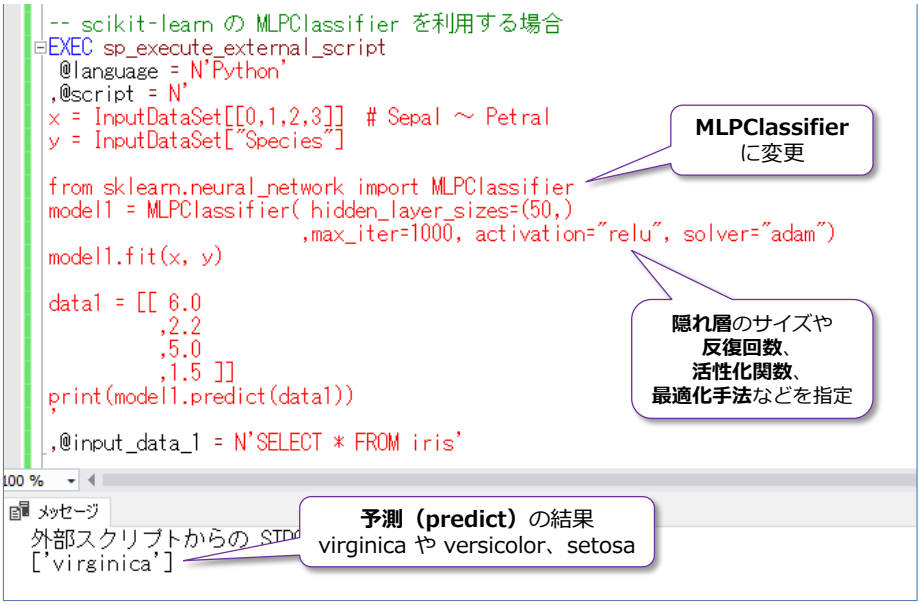
sklearn.neural_network.MLPClassifier
http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

MLPClassifier クラスも、これまでのクラスと利用方法はほとんど同じです。

```
-- scikit-learn の MLPClassifier を利用する場合
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
x = InputDataSet[[0,1,2,3]]  # Sepal ～ Petral
y = InputDataSet["Species"]
```

```
from sklearn.neural_network import MLPClassifier
model1 = MLPClassifier( hidden_layer_sizes=(50,)
                       ,max_iter=1000, activation="relu", solver="adam")
model1.fit(x, y)

data1 = [[ 6.0
          ,2.2
          ,5.0
          ,1.5 ]]
print(model1.predict(data1))'
,@input_data_1 = N'SELECT * FROM iris'
```

「**from sklearn.neural_network import MLPClassifier**」と「**model1 = MLPClassifier(~)**」のところを変更しているだけです。

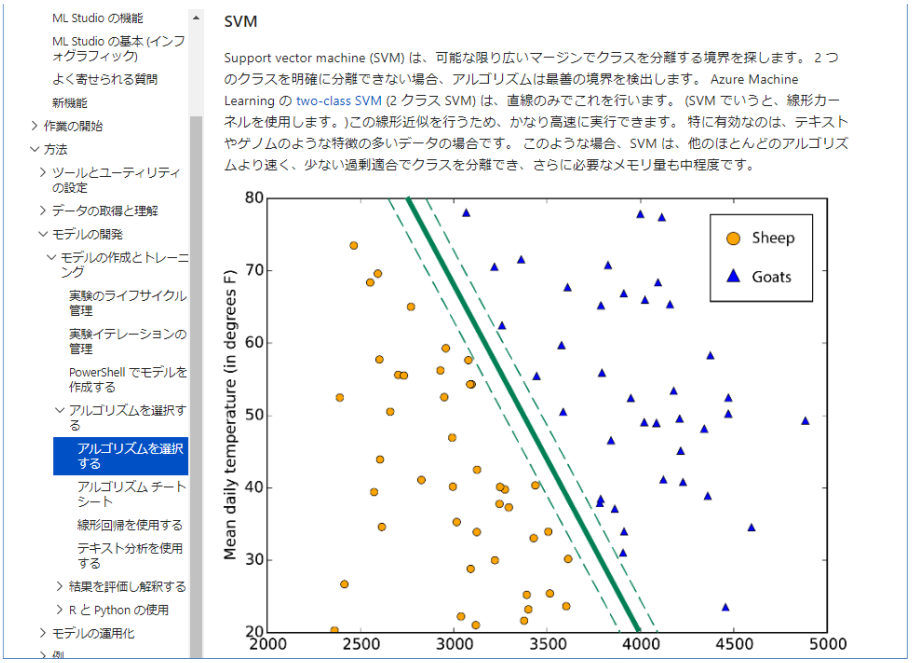


以上のように **scikit-learn**（**sklearn**）では、いろいろな機械学習のアルゴズムを利用できるので、Python を利用した機械学習ではよく利用されています。アルゴリズムの一覧は、**scikit-learn** サイトの以下のページで確認できます。

Choosing the right estimator
http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

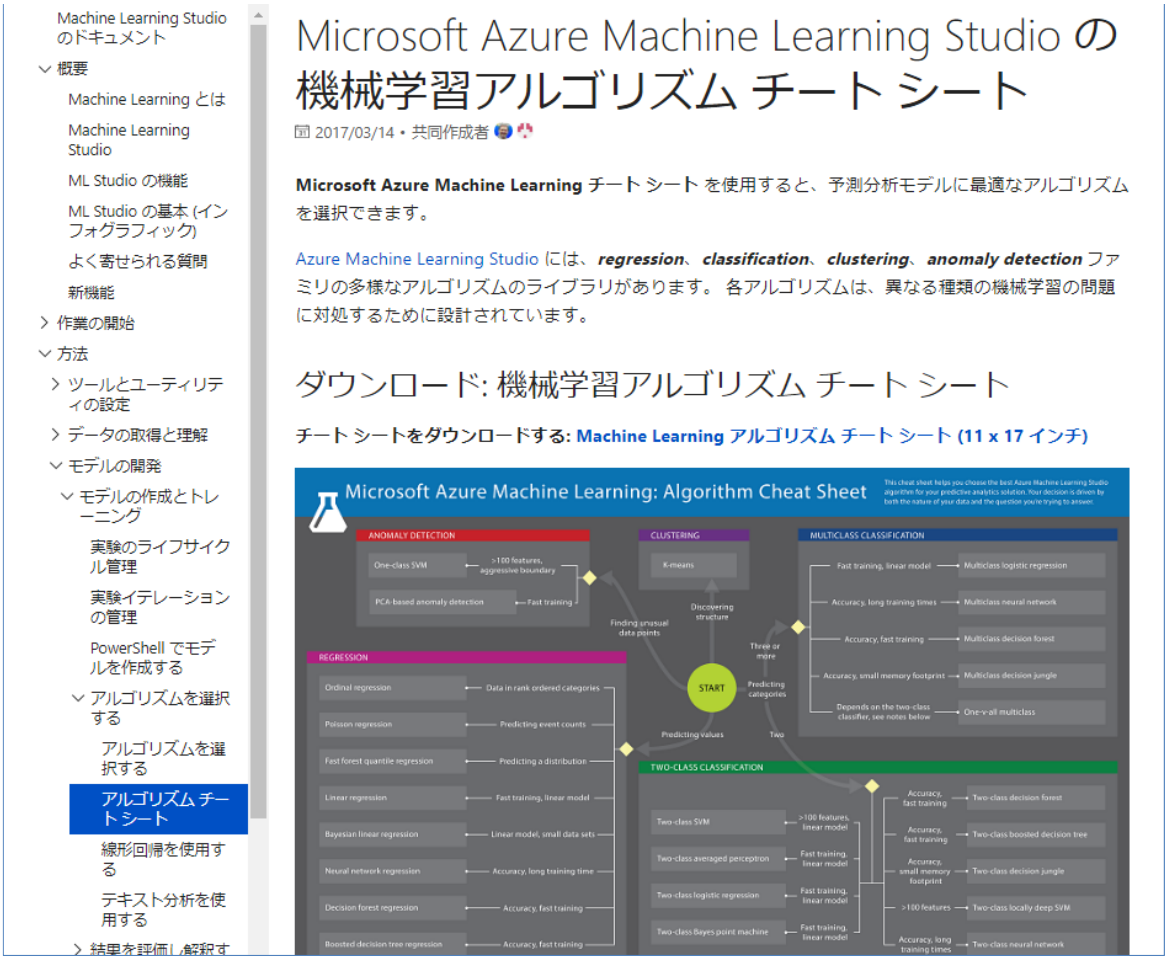
機械学習のアルゴリズムについては、**Azure ML**（Microsoft Azure Machine Learning）のヘルプになりますが、以下のヘルプが分かりやすくまとまっています。

Microsoft Azure Machine Learning のアルゴリズムの選択方法
https://docs.microsoft.com/ja-jp/azure/machine-learning/studio/algorithm-choice



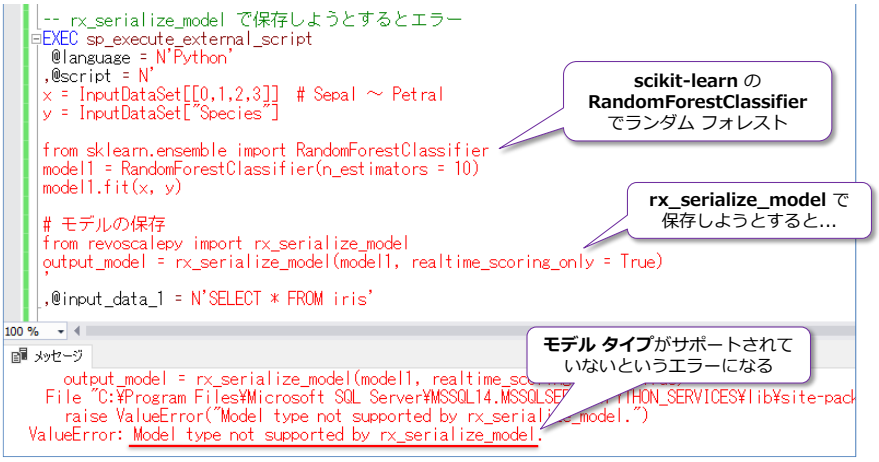
また、アルゴリズムの選択基準については、以下のチート シートが参考になります。

Microsoft Azure Machine Learning Studio の機械学習アルゴリズム チート シート
https://docs.microsoft.com/ja-jp/azure/machine-learning/studio/algorithm-cheat-sheet



3.5 pickle によるモデルの保存、予測結果の保存

モデルの保存については、2 章で Revoscalepy の **rx_dforest** で作成したランダム フォレストのモデルを **rx_serialize_model** でシリアライズ化して保存する方法を説明しました。しかし、**scikit-learn** の **RandomForestClassifier** や **LinearSVC**、**SGDClassifier** クラスで作成したモデルは、**rx_serialize_model** では保存することができません。保存しようとすると、次のようにエラーになります。

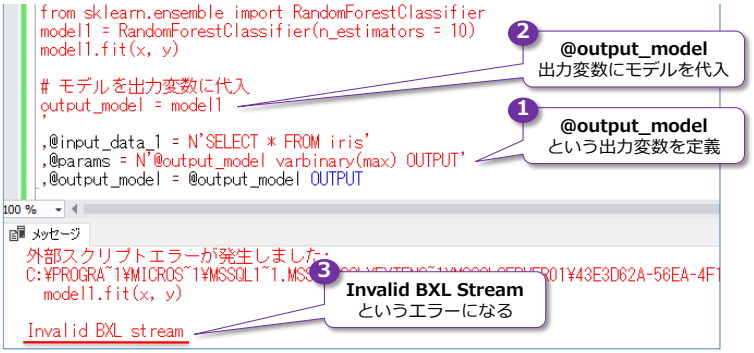


rx_serialize_model は、**Revoscalepy** での機械学習のアルゴリズムで作成したモデルを保存して、ネイティブ スコアリング（Transact-SQL の **PREDICT** 関数での予測）を行うためのものになっています。

ネイティブ スコアリングでサポートされているアルゴリズムには、**rx_dforest** や **rx_dtree**（決定木）、**rx_logit**（ロジスティック回帰）、**rx_lin_mod**（回帰モデル）などがありますが、詳しくは、SQL Server のヘルプの以下のページに記載されています。

ネイティブのスコアリング
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/sql-native-scoring

したがって、**scikit-learn** の **RandomForestClassifier** や **LinearSVC**、**SGDClassifier** クラスなど、Revoscalepy を利用していない場合には、モデルを保存する場合に、**pickle**（Python でオブジェクトを保存できるライブラリ）などを利用するようにします。なお、**pickle** を利用せずに、**出力変数**（**@params** パラメーターで **OUTPUT** で定義した変数）にモデルを出力しようとする場合は、次のようにエラーになります。



pickle でモデルを保存（pickle.dumps）

pickle は、Python では定番となっている、オブジェクトを保存することができるライブラリです。**pickle** では、「**pickle.dumps(オブジェクト名)**」のように **dumps** メソッドでオブジェクトを保存することができ、保存したものを取り出すときは「**pickle.loads(オブジェクト名)**」のように **loads** メソッドを利用します。

これも試してみましょう。ここでは、前項で **RandomForestClassifier** クラスを利用して作成したランダム フォレストのモデルを保存してみます。モデルの保存先は、2 章で利用した「**t_model**」テーブルにします。

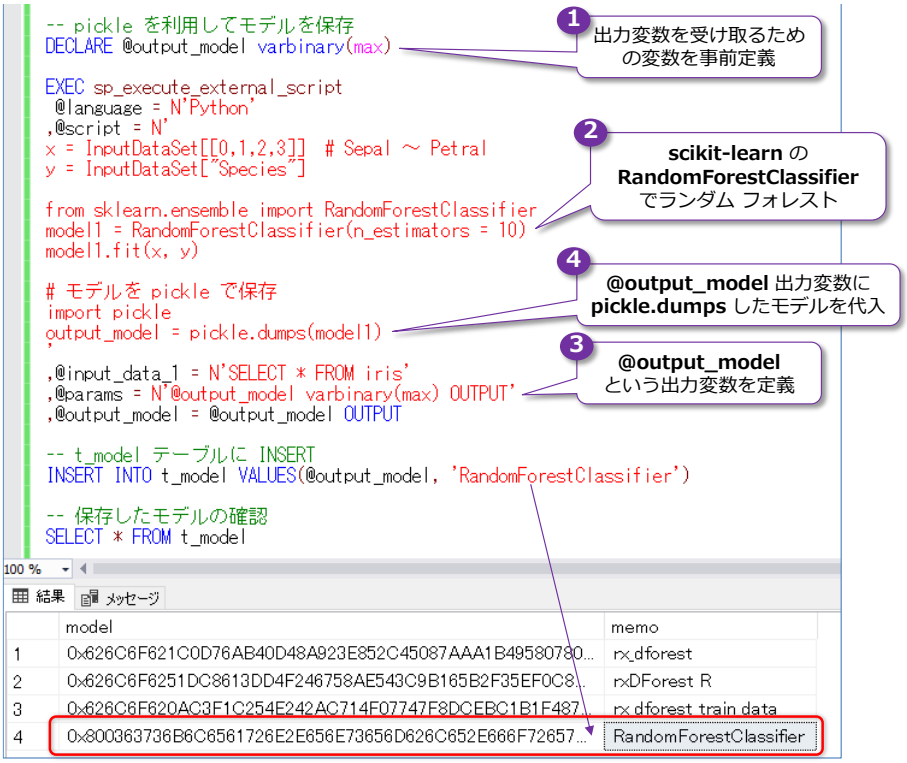
```
-- scikit-learn の RandomForestClassifier で作成したモデルを pickle で保存
DECLARE @output_model varbinary(max)

EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
x = InputDataSet[[0,1,2,3]]  # Sepal ～ Petral
y = InputDataSet["Species"]

from sklearn.ensemble import RandomForestClassifier
model1 = RandomForestClassifier(n_estimators = 10)
model1.fit(x, y)

# モデルを pickle で保存
import pickle
output_model = pickle.dumps(model1)
'
,@input_data_1 = N'SELECT * FROM iris'
,@params = N'@output_model varbinary(max) OUTPUT'
,@output_model = @output_model OUTPUT

-- t_model テーブルに INSERT
INSERT INTO t_model VALUES(@output_model, 'RandomForestClassifier')
```

pickle を利用するために「**import pickle**」を記述して、「**output_model = pickle.dumps (model1)**」でモデル（**model1**）を **output_model** 出力変数に代入しています。

出力変数の利用方法は、2 章での **rx_serialize_model** を利用した場合と同様で、**@params** で変数の定義、「**@output_model = @output_model OUTPUT**」で事前に定義した変数に値を代入して、それを **INSERT** ステートメントで **t_model** テーブルの **model** 列に格納しています。

INSERT が完了したら、保存したモデルを確認しておきましょう。

```
-- 保存したモデルの確認
SELECT * FROM t_model
```

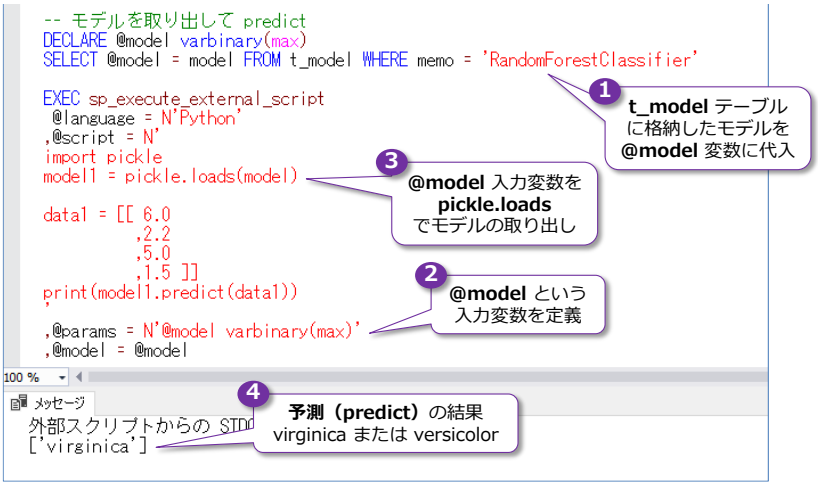


pickle でモデルの取り出し（pickle.loads）

次に、保存したモデルを取り出して、予測 (predict) を行ってみましょう。モデルを取り出すには、「**pickle.loads(オブジェクト名)**」を利用します。

```
-- モデルを取り出して predict
DECLARE @model varbinary(max)
SELECT @model = model FROM t_model WHERE memo = 'RandomForestClassifier'

EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
import pickle
model1 = pickle.loads(model)

data1 = [[ 6.0
          ,2.2
          ,5.0
          ,1.5 ]]
print(model1.predict(data1))
'
,@params = N'@model varbinary(max)'
,@model = @model
```

t_model テーブルから取り出したモデルを **@model** 変数に格納して、**@params** で定義した入力変数（**@model**）に与えています。この入力変数を **pickle.loads** メソッドで取り出せば、**predict** を実行できるようになります。

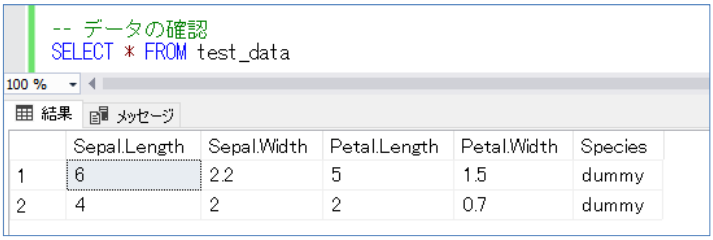


このように、Revoscalepy を利用していない場合でも、pickle を利用してモデルを保存することで、モデルの作成と予測を別々に実行できるようになります。

予測結果の保存

次に、予測結果を保存してみましょう。予測には、2 章で作成した「**test_data**」テーブルを利用します。

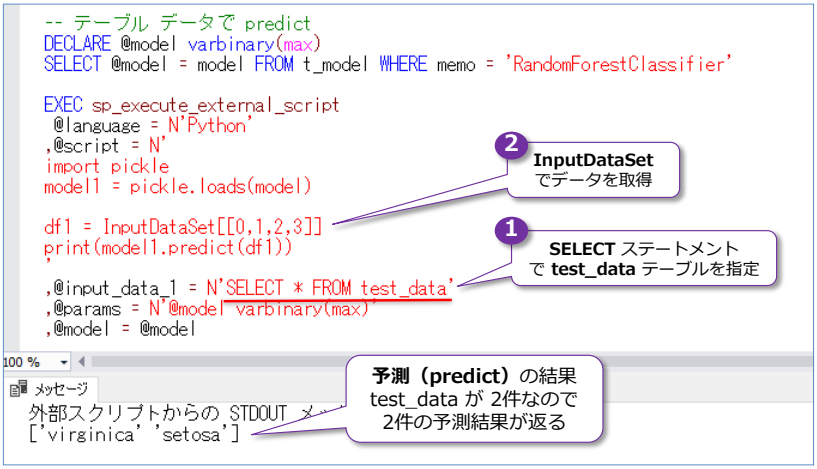
```
-- test_data テーブルのデータを確認
SELECT * FROM test_data
```



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	6	2.2	5	1.5	dummy
2	4	2	2	0.7	dummy

まずは、このテーブルで予測結果を確認してみます。

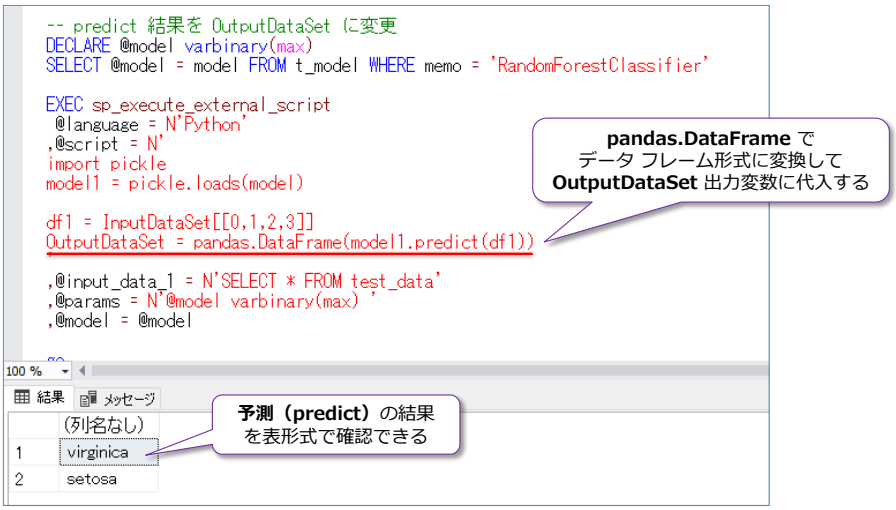
```
-- @input_data_1 に指定したテーブル データで predict
DECLARE @model varbinary(max)
SELECT @model = model FROM t_model WHERE memo = 'RandomForestClassifier'
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
import pickle
model1 = pickle.loads(model)

df1 = InputDataSet[[0,1,2,3]]
print(model1.predict(df1))
'
,@input_data_1 = N'SELECT * FROM test_data'
,@params = N'@model varbinary(max)'
,@model = @model
```



@input_data_1 で指定したデータを **InputDataSet** 入力変数で受け取って、「**InputDataSet[[0,1,2,3]]**」で 1 列目～4 列目の Sepal と Petal の Length／Width を取得して、**predict** の引数に与えています。

この例では、**predict** の結果を **print** で出力しているので、次に **OutputDataSet** 出力変数に代入して結果を表示してみます。**OutputDataSet** は、pandas の DataFrame 形式のデータを指定する必要があるので、次のように「**pandas.DataFrame(～)**」を利用して、形式を変換します。

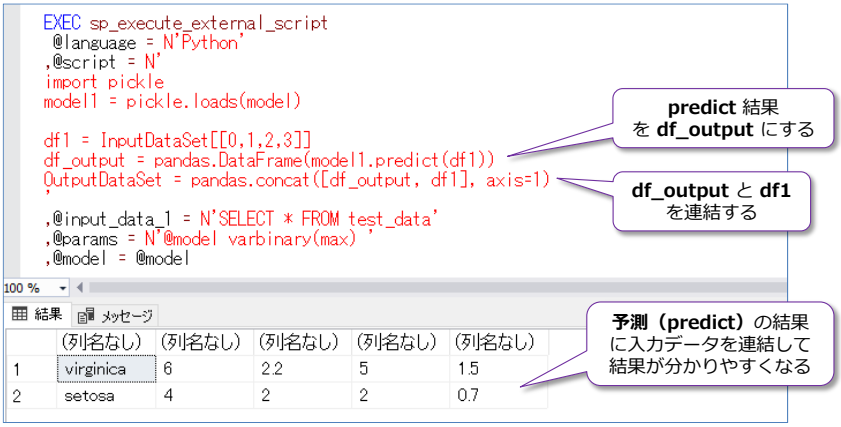
```
# predict 結果を OutputDataSet に変更
OutputDataSet = pandas.DataFrame(model1.predict(df1))
```



このように、データ フレーム形式に変換すると、予測結果が分かりやすくなります。

次に、予測結果だけでなく、予測に利用した入力データも合わせて表示してみましょう。これを行うには、pandas の **concat** メソッドを利用します。具体的には、**OutputDataSet** の部分を次のように変更します。

```
# predict 結果を df_output にして、入力データ (df1) と連結して OutputDataSet へ
df_output = pandas.DataFrame(model1.predict(df1))
OutputDataSet = pandas.concat([df_output, df1], axis=1)
```



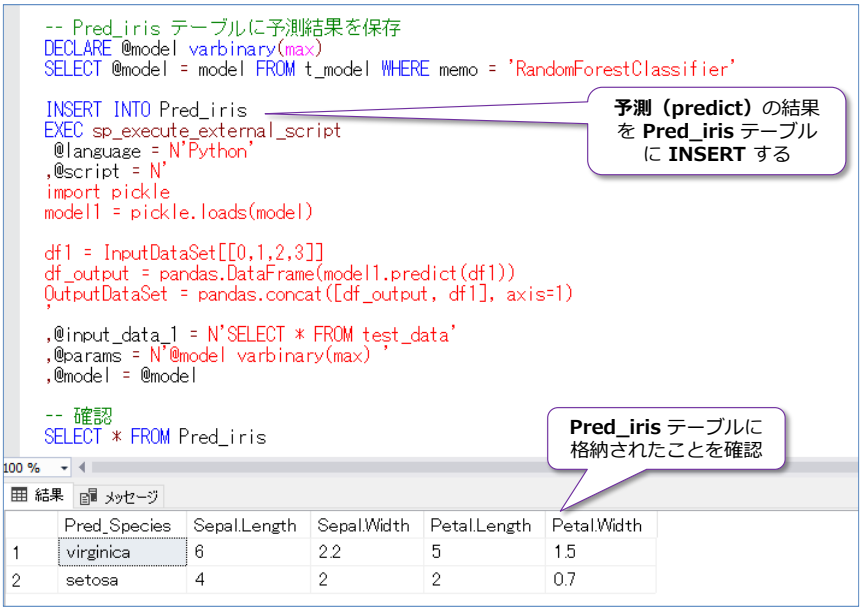
このように、予測結果に入力データを連結しておくと、結果が分かりやすくなります。

最後に、この予測結果をテーブルとして保存してみます。これを行うには、事前に格納先となるテーブルを次のように作成しておきます（列名は適当なものでもかまいません）。

```
-- 予測結果を格納するためのテーブルを作成しておく。「Pred_iris」という名前で作成
CREATE TABLE Pred_iris
( Pred_Species varchar(100)
 ,[Sepal.Length] float
 ,[Sepal.Width]  float
 ,[Petal.Length] float
 ,[Petal.Width]  float )
```

あとは、このテーブルに予測結果を格納するように「**INSERT INTO Pred_iris EXEC ～**」という形で **sp_execute_external_script** を実行すれば完成です。

```
INSERT INTO Pred_iris
EXEC sp_execute_external_script
       :
```



3.6 モデルの保存方法の違い（pickle vs. rx_serialize_model）

前項では、**scikit-learn** の **RandomForestClassifier** クラスで作成したモデルを **pickle** で保存する方法を説明しましたが、**Revoscalepy** の **rx_dforest**（ランダム フォレスト）や **rx_dtree**（決定木）、**rx_logit**（ロジスティック回帰）、**rx_lin_mod**（回帰モデル）などで作成したモデルに対しても **pickle** で保存することができます。もちろん、**rx_serialize_model** で保存したほうが性能が良いので、**pickle** での保存はお勧めではありませんが、ここでは、その性能差を確認するために、両方の保存方法を試してみましょう。

pickle vs. rx_serialize_model

まずは、**rx_dforest**（ランダム フォレスト）を利用して、モデルを作成して、これを **pickle** と **rx_serialize_model** の両方で保存してみましょう。

```
-- rx_dforest のモデルを pickle と rx_serialize_model の両方で保存
DECLARE @pickle_model varbinary(max), @output_model varbinary(max)

EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from revoscalepy import rx_dforest, rx_serialize_model
InputDataSet["Species"] = InputDataSet["Species"].astype("category")
model1 = rx_dforest("Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width"
                    ,data = InputDataSet, n_tree = 10)

# pickle でモデルを保存
import pickle
pickle_model = pickle.dumps(model1)

# rx_serialize_model でモデルを保存（ネイティブ モデル）
output_model = rx_serialize_model(model1, realtime_scoring_only = True)
'
,@input_data_1 = N'SELECT * FROM iris'
,@params = N'@pickle_model varbinary(max) OUTPUT
            ,@output_model varbinary(max) OUTPUT'
,@pickle_model = @pickle_model OUTPUT
,@output_model = @output_model OUTPUT

-- モデルを t_model テーブルに INSERT
INSERT INTO t_model VALUES(@pickle_model, 'rx_dforest pickle')
INSERT INTO t_model VALUES(@output_model, 'rx_dforest native')

-- 保存したモデルを確認
SELECT * FROM t_model
```

```
-- 保存したモデルを確認
SELECT * FROM t_model
```

100 %

結果 メッセージ

	model	memo
1	0x626C6F621EC5FFA2BD38A3305DBAF1DE924E5087AB0EA26...	rx_dforest
2	0x626C6F6296E2EE57B3B7E46886B8D693731C90B1012B544C...	rxDForest R
3	0x626C6F627D45666803D066CA95A30B5C636F419445C192CF...	rx_dforest train_data
4	0x800363736B6C6561726E2E656E73656D626C652E666F72657...	RandomForestClassifier
5	0x8003637265766F7363616C6570792E66756E6374696F6E732E...	rx_dforest pickle
6	0x626C6F62350A6AF16398E3FBEC2523A6BBABADB70632597...	rx_dforest native

次にモデルのサイズを比較してみます（**DATALENGTH** 関数でデータ サイズを確認します）。

```
-- モデルのサイズを確認
SELECT DATALENGTH(model), * FROM t_model
```

```
-- モデルのサイズを確認
SELECT DATALENGTH(model), * FROM t_model
```

100 %

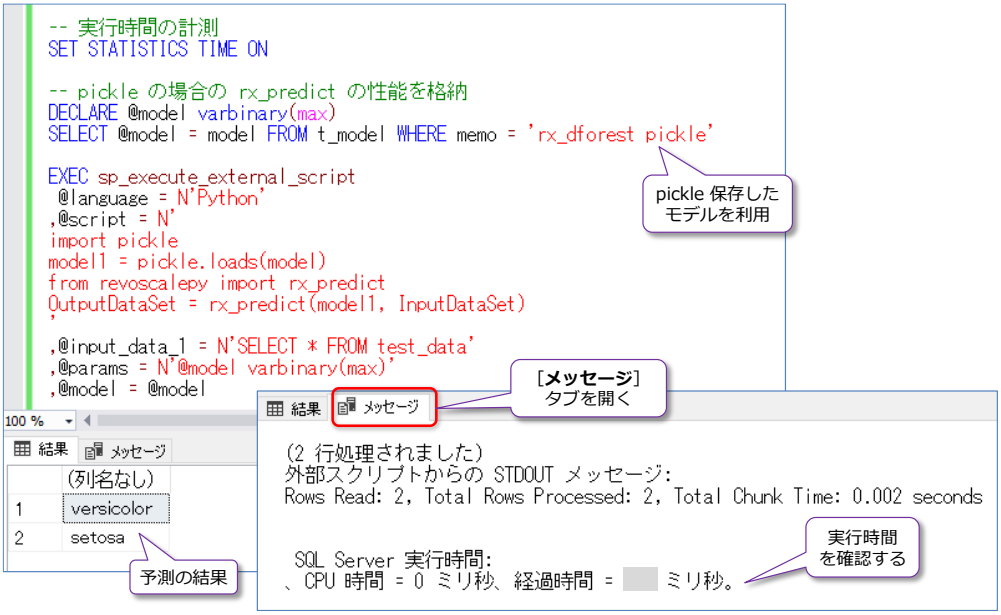
結果 メッセージ

	(列名なし)	model	memo
1	21912	0x626C6F621EC5FFA2BD38A3305DBAF1DE924E5087AB0EA26...	rx_dforest
2	21869	0x626C6F6296E2EE57B3B7E46886B8D693731C90B1012B544C...	rxDForest R
3	20910	0x626C6F627D45666803D066CA95A30B5C636F419445C192CF...	rx_dforest train_data
4	17623	0x800363736B6C6561726E2E656E73656D626C652E666F72657...	RandomForestClassifier
5	54280	0x8003637265766F7363616C6570792E66756E6374696F6E732E...	rx_dforest pickle
6	21154	0x626C6F62350A6AF16398E3FBEC2523A6BBABADB70632597...	rx_dforest native

pickle で保存したモデルよりも、**rx_serialize_model** で保存したモデルのほうが、半分以下のサイズになっていることを確認できます。

次に、予測の実行時間を比較してみましょう。これを行うには、「**SET STATISTICS TIME ON**」コマンドを実行して、実行時間を計測するようにします。最初は、**pickle** で保存したものを利用して、**rx_predict** で予測を実行してみます（入力データには **test_data** テーブルを利用）。

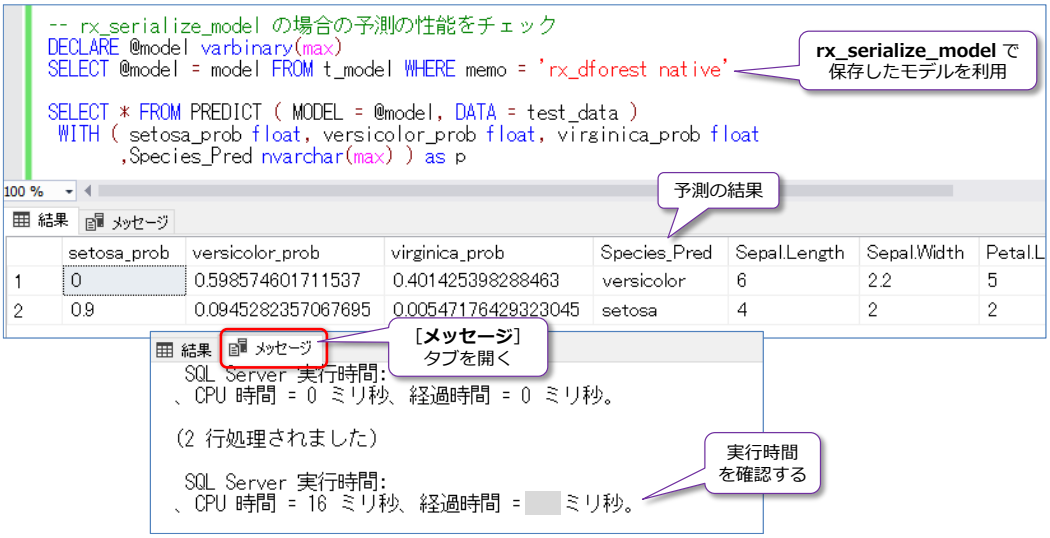
```
-- 実行時間の計測
SET STATISTICS TIME ON

-- pickle の場合の rx_predict の性能をチェック
DECLARE @model varbinary(max)
SELECT @model = model FROM t_model WHERE memo = 'rx_dforest pickle'

EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
import pickle
model1 = pickle.loads(model)
from revoscalepy import rx_predict
OutputDataSet = rx_predict(model1, InputDataSet)
'
,@input_data_1 = N'SELECT * FROM test_data'
,@params = N'@model varbinary(max)'
,@model = @model
```



実行後、[**メッセージ**] タブを開くと、「**SET STATISTICS TIME ON**」コマンドで計測した実行時間が「**経過時間**」という形で確認できるので、これをメモしておきます。

次に、**rx_serialize_model** で保存したモデルを利用して、ネイティブ スコアリング（PREDICT 関数）で予測を実行してみます。

```
-- rx_serialize_model の場合の予測の性能をチェック
DECLARE @model varbinary(max)
SELECT @model = model FROM t_model WHERE memo = 'rx_dforest native'

SELECT * FROM PREDICT ( MODEL = @model, DATA = test_data )
 WITH ( setosa_prob float, versicolor_prob float, virginica_prob float
       ,Species_Pred nvarchar(max) ) as p
```

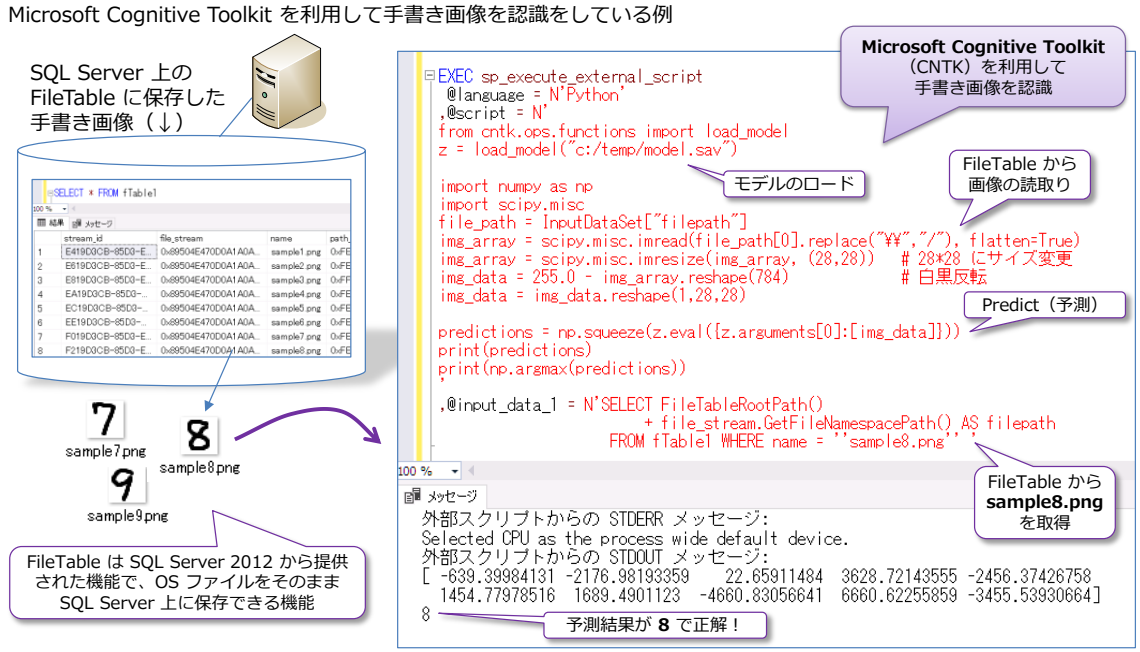


pickle で保存したモデルを利用した予測よりも、速く実行できていることを確認できると思います。このように Revoscalepy は、性能面で大きなメリットがあります。

3.7 Microsoft Cognitive Toolkit（CNTK）を利用した画像認識

ここでは、**ディープ ラーニング**（深層学習）のフレームワークである **Microsoft Cognitive Toolkit**（CNTK）を利用して、画像を認識する方法を説明します。次の図のように、SQL Server 上の FileTable に手書き画像を保存して、それを Microsoft Cognitive Toolkit で作成した画像認識モデルを利用して、予測してみます。

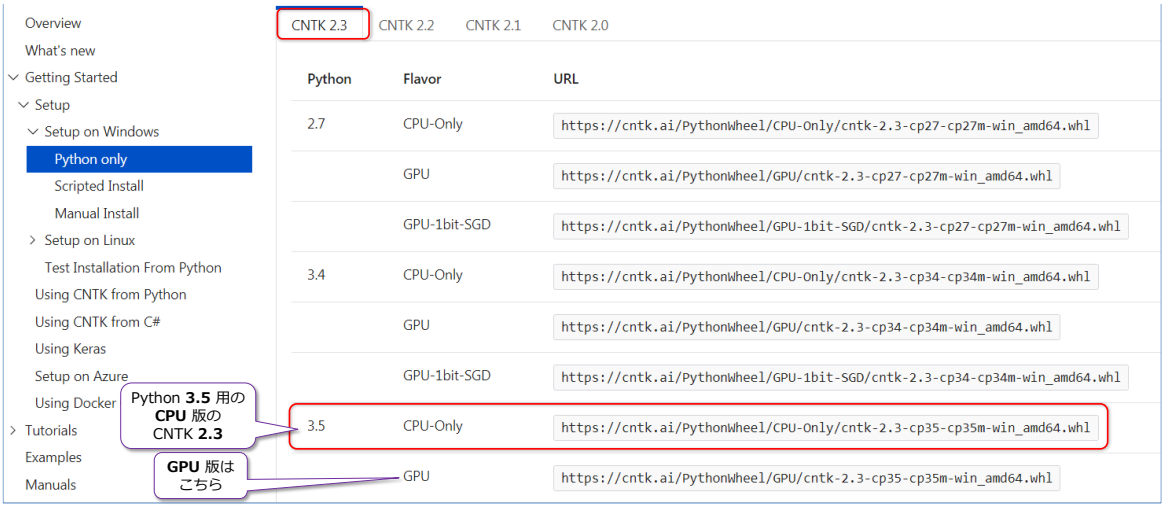
Microsoft Cognitive Toolkit を利用して手書き画像を認識をしている例



Microsoft Cognitive Toolkit のインストール（CNTK 2.3 の場合）

Microsoft Cognitive Toolkit のインストール方法は、次の URL に記載されています。

Installing CNTK for Python on Windows
https://docs.microsoft.com/en-us/cognitive-toolkit/setup-windows-python

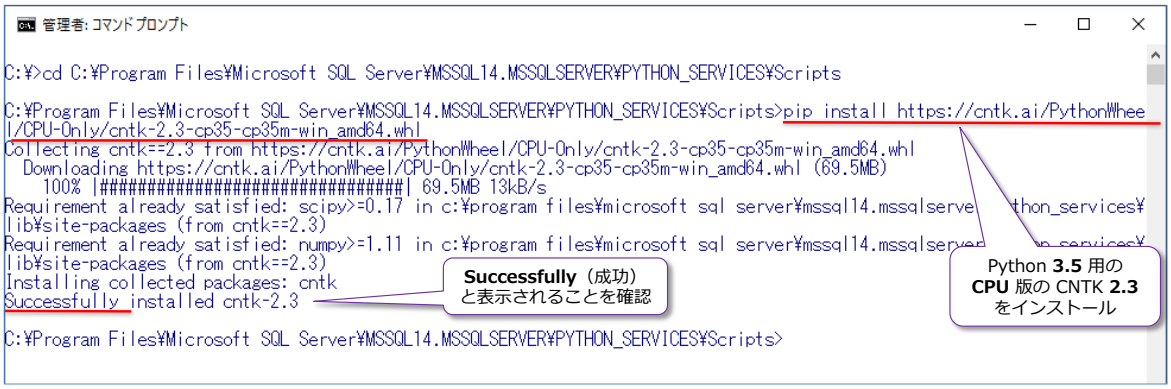


このページは、Microsoft Cognitive Toolkit（CNTK）のバージョンごとにタブがあり、執筆時点

（2017 年 11 月時点）では **CNTK 2.3** が最新バージョンです。「**CNTK 2.3**」タブでは、Python のバージョンごと、**CPU-Only** なのか **GPU** 版なのかで URL が記載されています。SQL Server 2017 の Machine Learning Services は、**Python 3.5.2** を利用しているので、**Python 3.5** の **CPU-Only** のところが必要な URL になります（GPU を搭載しているマシンの場合は、GPU 版の URL を利用することもできます）。

この URL を「**pip install URL**」という形で利用すれば、**CNTK 2.3** をインストールすることができます（コマンド プロンプトを［**管理者として実行**］で開いて、次のようにコマンドを実行します）、

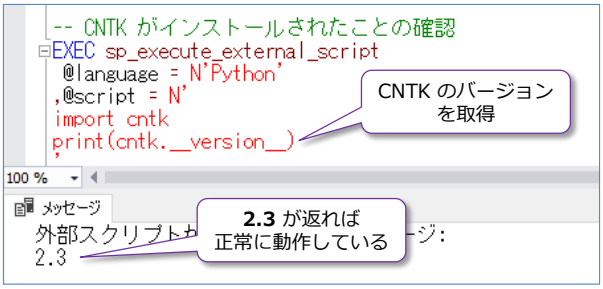
```
cd C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES¥Scripts
pip install https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp35-cp35m-win_amd64.whl
```



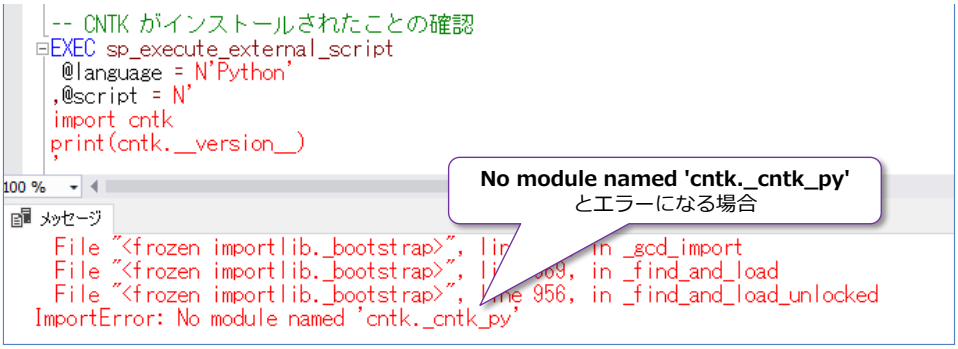
なお、SQL Server 2017 を名前付きインスタンスとしてインストールしている場合は、**MSSQLSERVER** の部分をインスタンス名に変更する必要があります。

インストール後は、CNTK が動作することを確認するために、クエリ エディターから sp_execute_external_script を次のように実行します。

```
-- CNTK がインストールされたことの確認
EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
import cntk
print(cntk.__version__)'
```



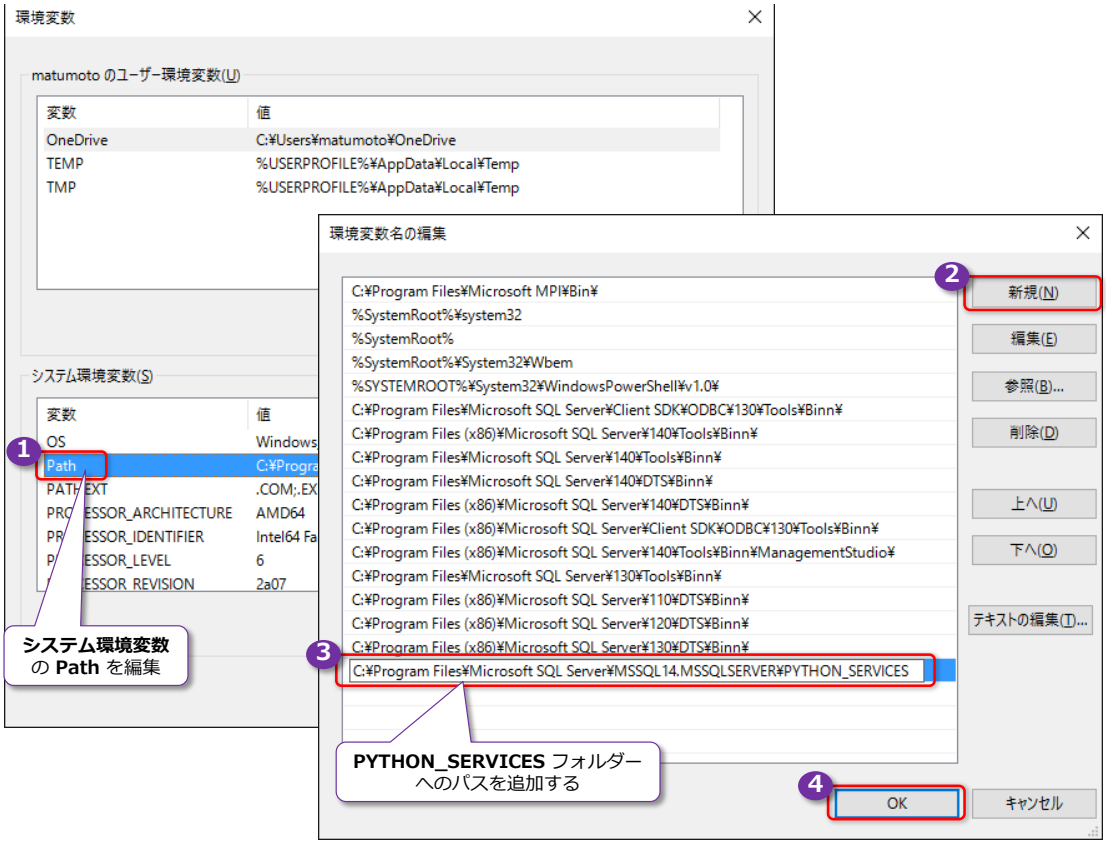
「**cntk.__version__**」で CNTK のバージョンを取得することができるので、これで「**2.3**」が返ってくれば、CNTK が正常に動作しています。もし、次のように「**No module named 'cntk._cntk_py'**」エラーが返る場合は、後述の追加の手順が必要になります。



このエラーが出る場合は、以下のパスを、**システム環境変数**の **PATH** に追加します（SQL Server 2017 を既定のインスタンスとしてインストールしている場合）。

C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES

名前付きインスタンスとしてインストールしている場合は、**MSSQLSERVER** の部分をインスタンス名に変更する必要があります。



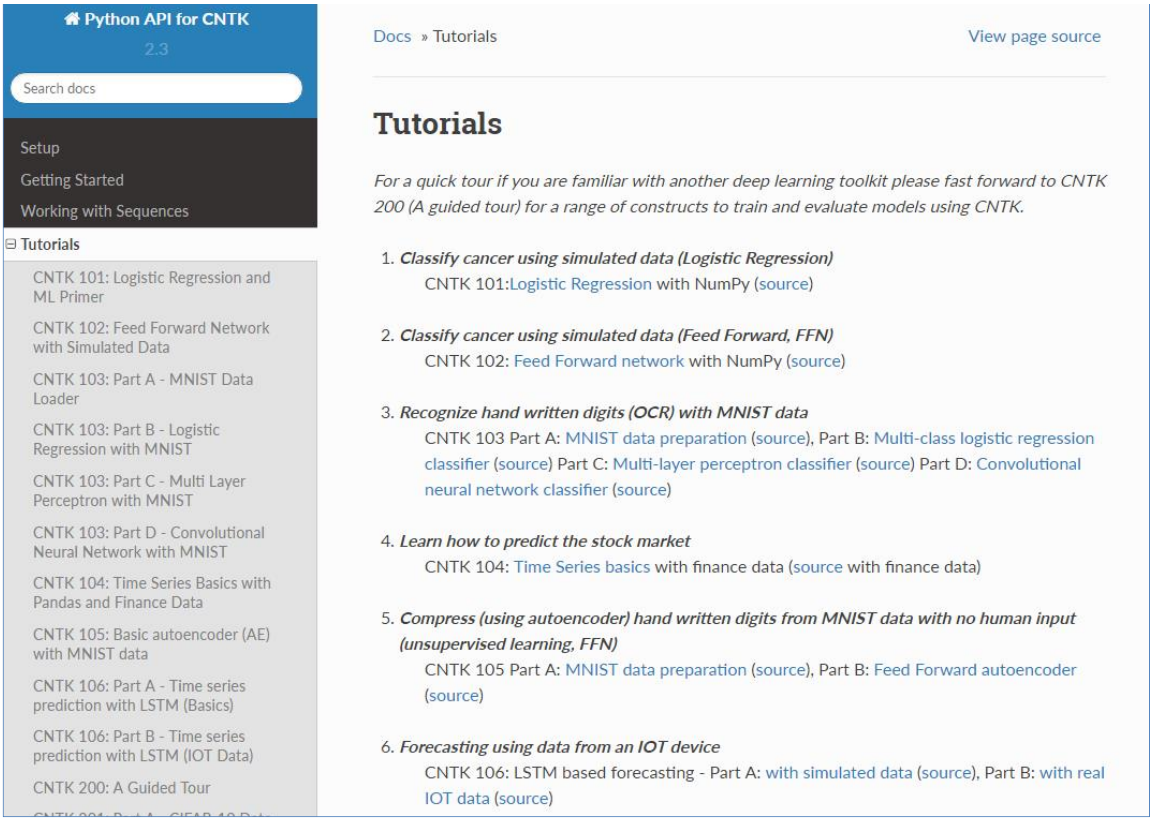
パスを追加した後は、SQL Server サービスを再起動します。それでも同様のエラーが出る場合は、OS を再起動してみたり、「**pip uninstall cntk**」と実行して、いったん CNTK を削除してから、再度 CNTK のインストールを実行してみてください。

Microsoft Cognitive Toolkit のチュートリアル

Microsoft Cognitive Toolkit はチュートリアル サイトが非常に充実しているので、まずは以下のチュートリアル サイトにアクセスすることをお勧めします。

Getting started
https://cntk.ai/pythondocs/gettingstarted.html

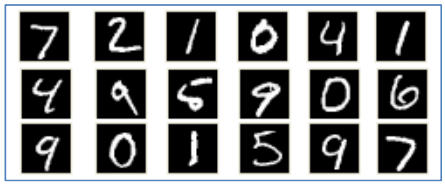
Tutorials
https://cntk.ai/pythondocs/tutorials.html



Microsoft Cognitive Toolkit で画像認識モデルの作成（MNIST の手書き数字）

次に、ディープ ラーニング（深層学習）での Hello World（最初の一歩）として有名な「**MNIST の手書き数字**」を利用して、画像認識のモデルを作成してみましょう。MNIST のデータは、http://yann.lecun.com/exdb/mnist で提供されていて、次のような手書きの数字データがダウンロードできるようになっています。

MNIST の手書き数字の例



トレーニング（訓練）用のデータとして 60,000 個、テスト用のデータとして 10,000 個の手書き

数字があり、それぞれ **.gz** 形式の圧縮ファイルとしてダウンロードできます。Microsoft Cognitive Toolkit のチュートリアルの 3 番目の「**Recognize hand written digits (OCR) with MNIST data**」では、ファイルをダウンロードして、それをもとに手書き画像を認識するモデルを作成することができるものが用意されているので（解説も付いているので）、ここではそれをそのまま利用します。

Microsoft Cognitive Toolkit のチュートリアルの 3番目（↓）

> 3. *Recognize hand written digits (OCR) with MNIST data*
>
> CNTK 103 Part A: MNIST data preparation (source), Part B: Multi-class logistic regression classifier (source) Part C: Multi-layer perceptron classifier (source) Part D: Convolutional neural network classifier (source)

このチュートリアルは **Part A** ～ **Part D** の 4 つに分かれていて、順番に試すのがお勧めですが、Part A の「**MNIST data preparation**」は、Part B～D を試すために必須の作業になり、MNIST のデータ（**.gz**）をダウンロードして、Part B 以降の手順を試すためのデータ加工（手書き数字データをテキスト ファイルとして保存する加工）を行っています。Part B と C は省略可能で、Part D の「**Convolutional neural network classifier**」を試すこともできます。

Convolutional neural network（CNN：畳み込みニューラル ネットワーク）は、画像認識や音声認識で非常によく利用されているニューラル ネットワークで、現在のディープ ラーニングは、ほとんどが CNN がベースになっています。

Part A および Part D は、SQL Server 2017 の sp_execute_external_script の **@script** に記述して試すこともできるのですが、これだとカラーリング（コードの色分け）が効かないので、Python の開発環境としてよく利用される「**Jupyter Notebook**」や「**PyCharm**」、「**Visual Studio Code**」などを利用して試すのがお勧めです。

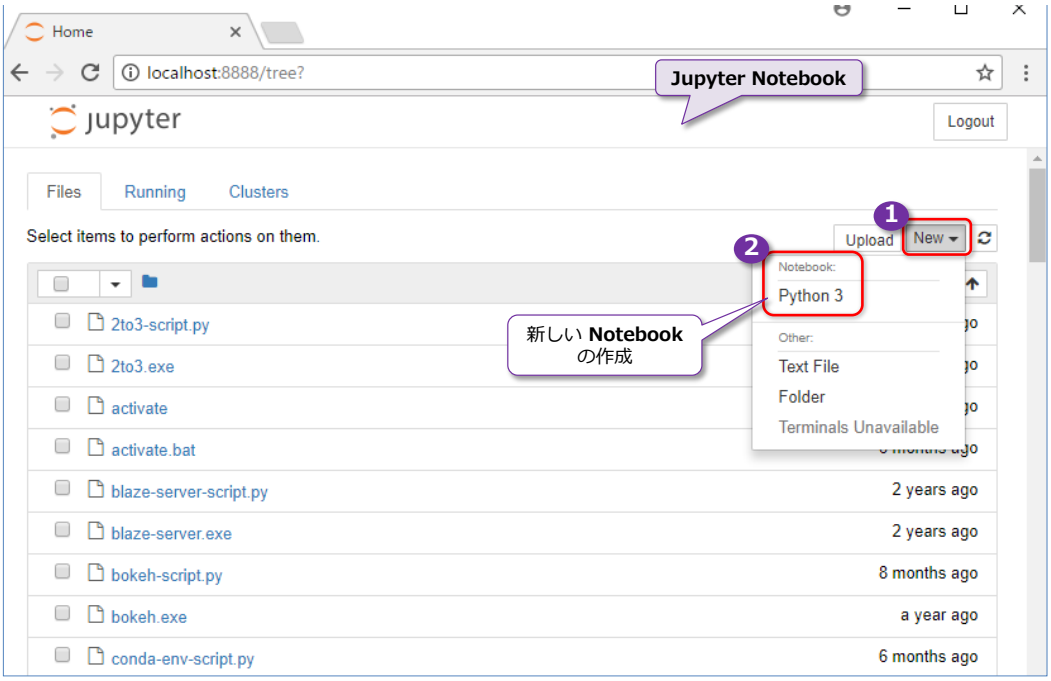
Jupyter Notebook を利用する場合は、pip.exe や conda.exe と同じフォルダー（**PYTHON_SERVICES** の **Scripts** フォルダー）にインストールされているので、次のようにコマンド プロンプトで「**jupyter notebook**」と記述すれば、起動することができます。

```
cd C:¥Program Files¥Microsoft SQL Server¥MSSQL14.MSSQLSERVER¥PYTHON_SERVICES¥Scripts
jupyter notebook
```

なお、SQL Server 2017 を名前付きインスタンスとしてインストールしている場合は、**MSSQLSERVER** の部分をインスタンス名に変更する必要があります。

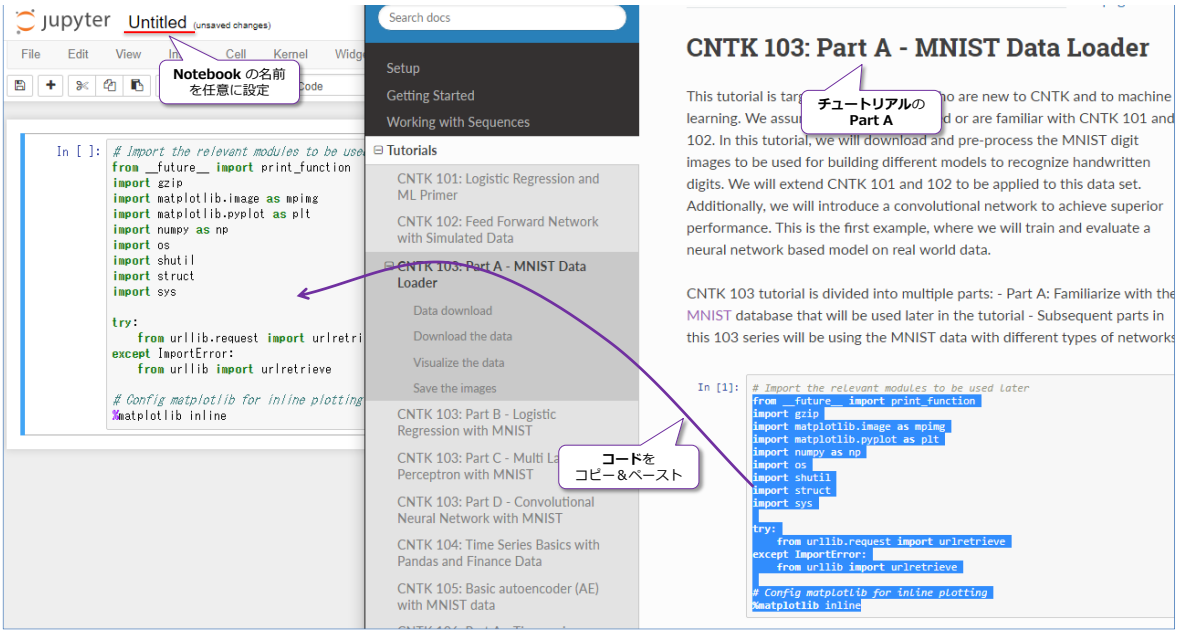
Jupyter Notebook が起動すると、次のように Web ブラウザーが起動して、ホーム ページが表

示されます（もし Web ブラウザーが起動しない場合は、手動で起動して、http://localhost:8888 にアクセスしてみてください）。

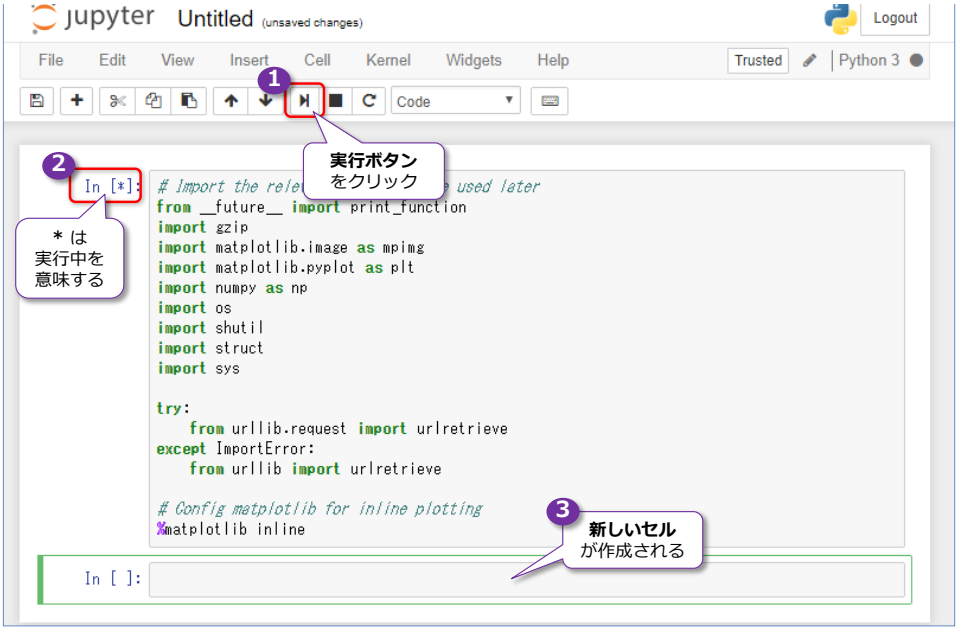


Jupyter Notebook では、Notebook という単位で Python スクリプトを管理しているので、まずは画面右側の「**New**」ボタンから「**Python 3**」をクリックして、新しい Notebook を作成します。

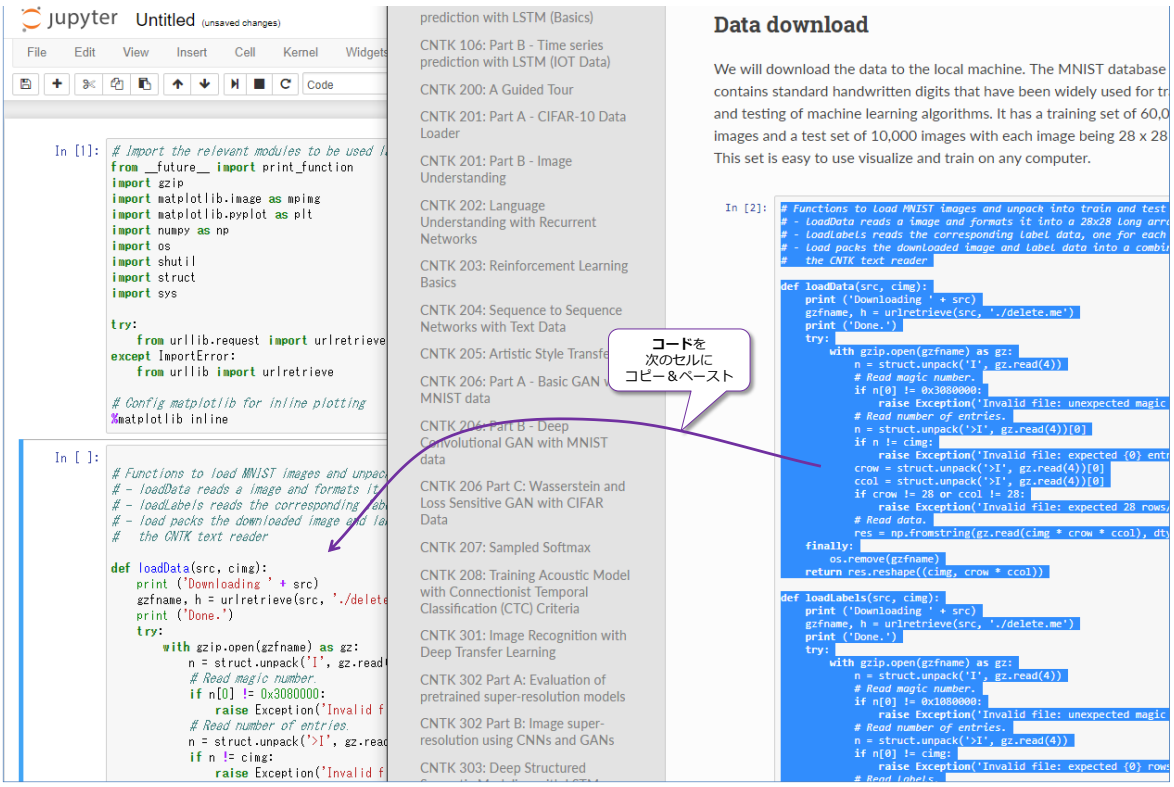
新しい Notebook が開いたら、次のようにチュートリアル サイトからコードをコピー＆ペーストします。



コードを貼り付けたら、次のように Notebook の［>］（実行ボタン）をクリックします。



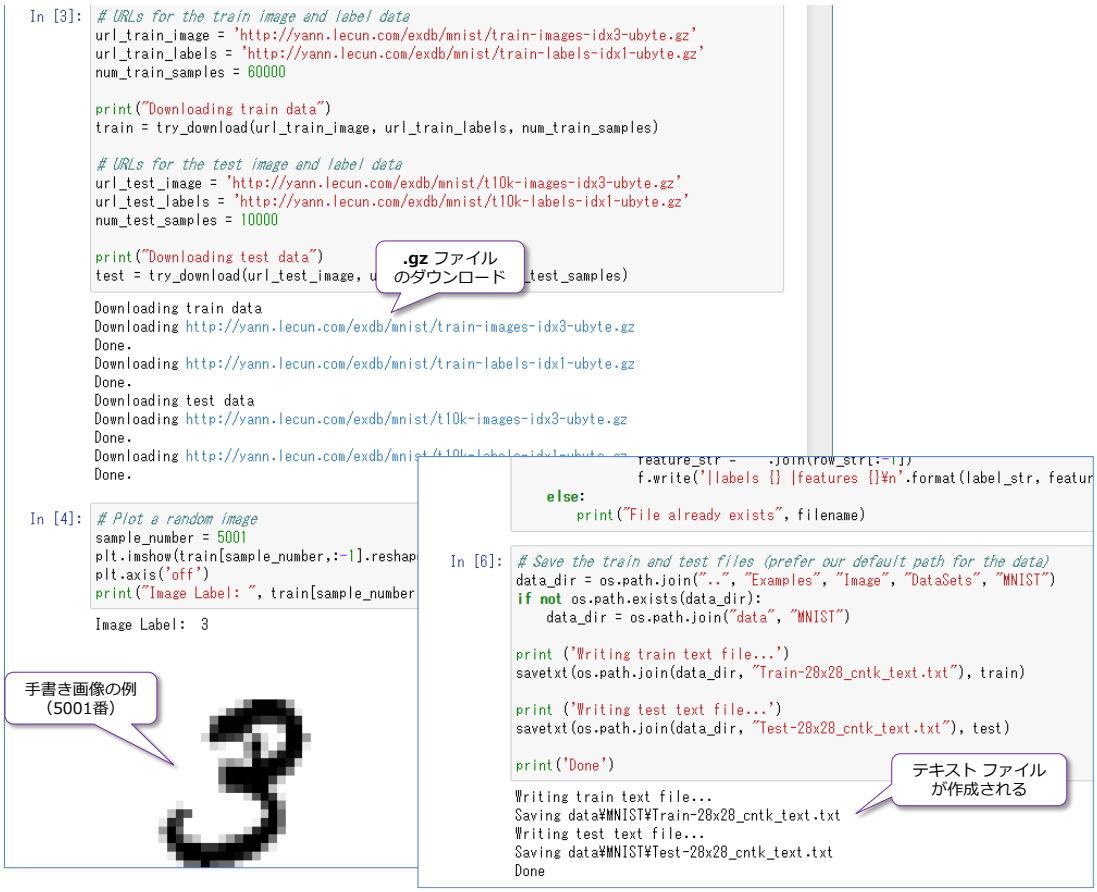
実行中は「**In [*]**」のように「*」が表示されて、実行が完了すると「**In [1]**」に変わって、実行回数がインクリメントされます。また、実行によって新しいセルが作成されるので、次のコードは、その新しいセルに貼り付けて実行していけます。



チュートリアルも Jupyter Notebook をベースに解説があるので、チュートリアル内に「**In [x]**」という部分があったら、そこのコードを新しいセルに貼り付けて実行、という形でチュートリアルを進めていくことができます。

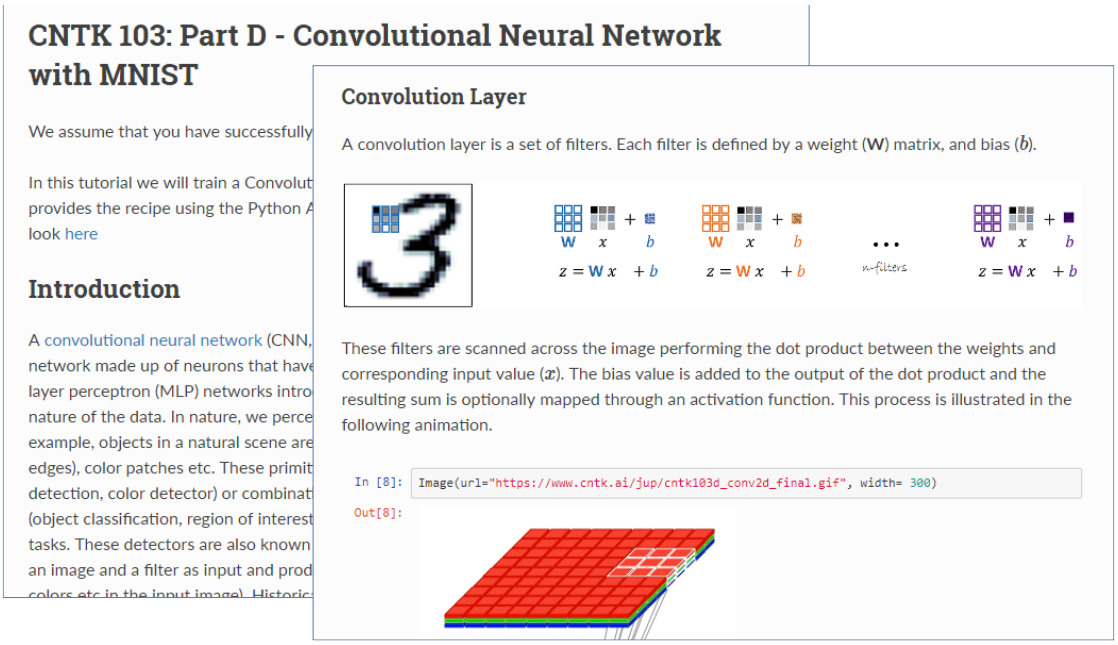
Part A のすべてのコードを実行すると、MNIST の 6 万点のトレーニング データと 1 万点のテスト データのダウンロード、およびデータの加工（Part B 以降のために、手書き画像データのテ

キスト ファイルへの変換）を行うことができます。

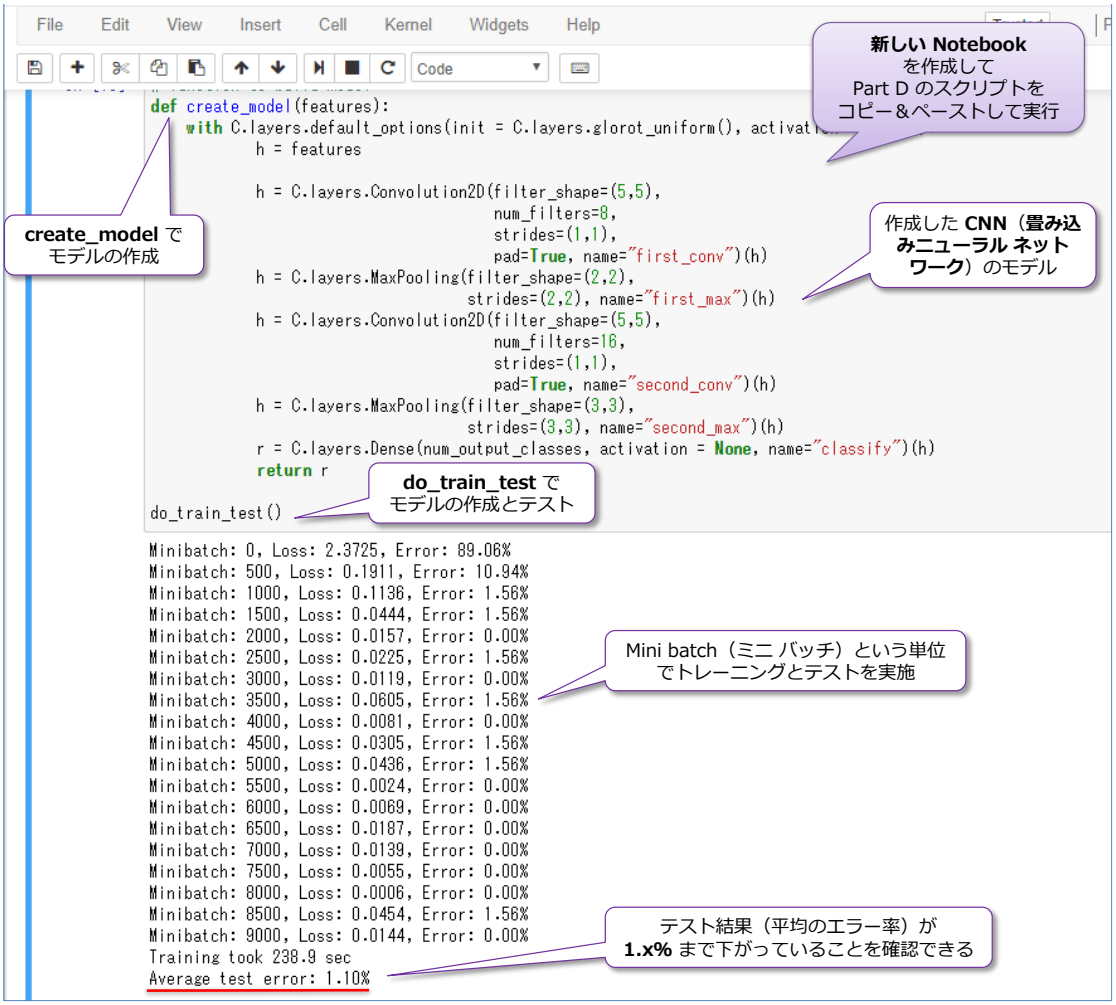


以上、Part A が完了した後は、Part B、C と進めていくのがお勧めですが、Part D の **CNN（畳み込みニューラル ネットワーク）**をすぐに試すこともできます。

CNTK 103: **Part D** - **Convolutional Neural Network** with MNIST
https://cntk.ai/pythondocs/CNTK_103D_MNIST_ConvolutionalNeuralNetwork.html



Part D のコードは、新しい Notebook を作成したほうが管理しやすくなるので、「**File**」メニューから「**New Notebook**」をクリックするか、Jupyter Notebook の起動時に表示されたホーム ページから「**New**」ボタンをクリックして新しい Notebook を作成しておくことをお勧めします。



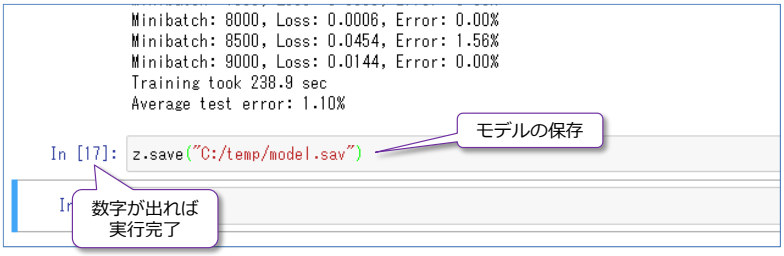

```
def create_model(features):
    with C.layers.default_options(init = C.layers.glorot_uniform(), activat
        h = features

        h = C.layers.Convolution2D(filter_shape=(5,5),
                                   num_filters=8,
                                   strides=(1,1),
                                   pad=True, name="first_conv")(h)
        h = C.layers.MaxPooling(filter_shape=(2,2),
                                strides=(2,2), name="first_max")(h)
        h = C.layers.Convolution2D(filter_shape=(5,5),
                                   num_filters=16,
                                   strides=(1,1),
                                   pad=True, name="second_conv")(h)
        h = C.layers.MaxPooling(filter_shape=(3,3),
                                strides=(3,3), name="second_max")(h)
        r = C.layers.Dense(num_output_classes, activation = None, name="classify")(h)
        return r

do_train_test()
```

```
Minibatch: 0, Loss: 2.3725, Error: 89.06%
Minibatch: 500, Loss: 0.1911, Error: 10.94%
Minibatch: 1000, Loss: 0.1136, Error: 1.56%
Minibatch: 1500, Loss: 0.0444, Error: 1.56%
Minibatch: 2000, Loss: 0.0157, Error: 0.00%
Minibatch: 2500, Loss: 0.0225, Error: 1.56%
Minibatch: 3000, Loss: 0.0119, Error: 0.00%
Minibatch: 3500, Loss: 0.0805, Error: 1.56%
Minibatch: 4000, Loss: 0.0081, Error: 0.00%
Minibatch: 4500, Loss: 0.0305, Error: 1.56%
Minibatch: 5000, Loss: 0.0436, Error: 1.56%
Minibatch: 5500, Loss: 0.0024, Error: 0.00%
Minibatch: 6000, Loss: 0.0069, Error: 0.00%
Minibatch: 6500, Loss: 0.0187, Error: 0.00%
Minibatch: 7000, Loss: 0.0139, Error: 0.00%
Minibatch: 7500, Loss: 0.0055, Error: 0.00%
Minibatch: 8000, Loss: 0.0006, Error: 0.00%
Minibatch: 8500, Loss: 0.0454, Error: 1.56%
Minibatch: 9000, Loss: 0.0144, Error: 0.00%
Training took 238.9 sec
Average test error: 1.10%
```

モデルの保存（save）

Part D の最後のコード（**do_train_test** メソッド）の実行が完了すると、CNN（畳み込みニューラル ネットワーク）のモデルが「**z**」という名前で完成しています。このモデルは、次のように **save** メソッドでファイルとして保存することができるので、これを実行しておいてください。

```
z.save("C:/temp/model.sav")
```

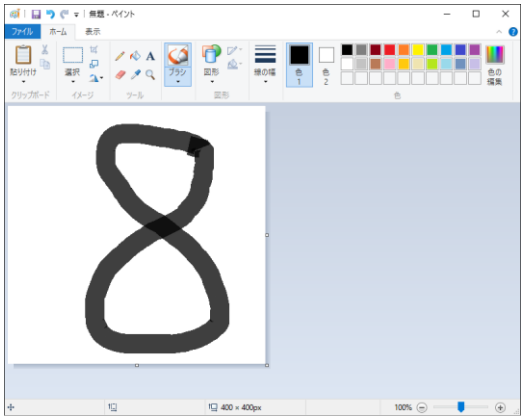


ファイルのパスには「**C:¥temp**」フォルダーを指定していますが、皆さんの環境に合わせて、任意の場所に変更してください。また、パスの指定に「**¥**」マークではなく「**/**」（スラッシュ）を利用していますが、Python ではパスの「**/**」を「**¥**」として解釈してくれるので、このように記述しています。もし、「¥」をパスに利用する場合は、「**"C:¥¥temp¥¥model.sav"**」のように「**¥¥**」（¥ マークを 2 個）にする必要があります（∵Python では ¥ マークがエスケープ文字のため）。

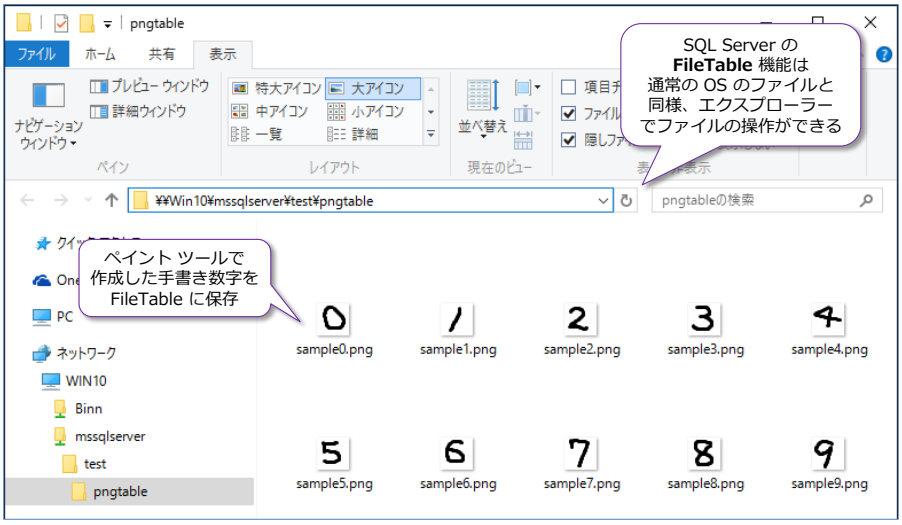
保存したモデルを SQL Server 2017 から利用（load_model、eval）

次に、SQL Server 2017 の Machine Learning Services を利用して、CNN のモデルを呼び出して（**load_model** メソッドでロードして）、手書きの画像を**予測**（**eval** メソッドで評価）してみます。手書きの画像は、**mspaint**（ペイント ツール）などで任意に作成して、SQL Server 上の **FileTable** に保存した場合を想定しています（FileTable の利用方法は、サンプル スクリプトの「**3-2_FileTable.sql**」ファイルに記載しています）。

ペイント ツールで手書きの数字を作成（↓）



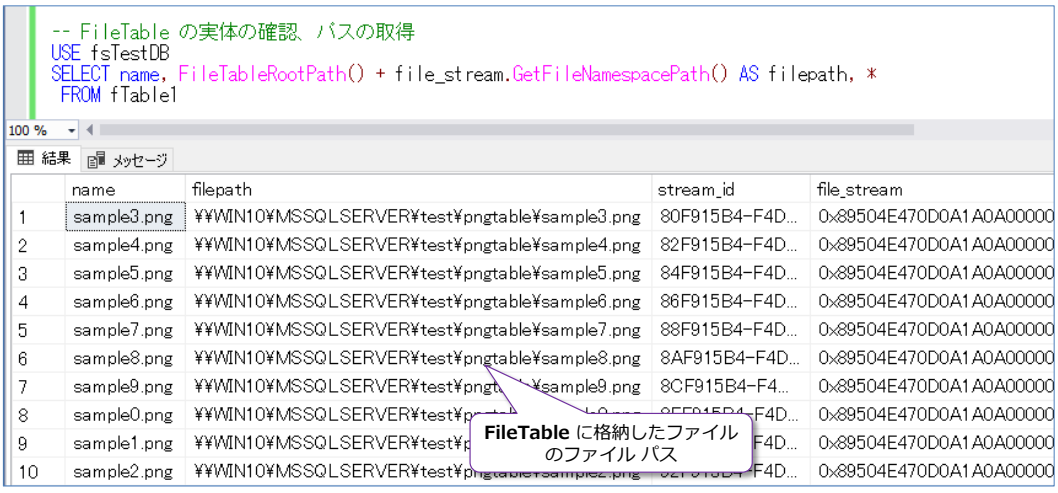
手書きの数字を FileTable に保存（↓）



FileTable に格納したファイルのパスは、次のように「**FileTableRootPath**」および「**GetFileNamespacePath**」関数で取得することができます。

```
-- FileTable の実体の確認、パスの取得
USE fsTestDB
```

```
SELECT name, FileTableRootPath() + file_stream.GetFileNamespacePath() AS filepath, *
 FROM fTable1
```



	name	filepath	stream_id	file_stream
1	sample3.png	¥¥WIN10¥MSSQLSERVER¥test¥pngtable¥sample3.png	80F915B4-F4D...	0x89504E470D0A1A0A00000
2	sample4.png	¥¥WIN10¥MSSQLSERVER¥test¥pngtable¥sample4.png	82F915B4-F4D...	0x89504E470D0A1A0A00000
3	sample5.png	¥¥WIN10¥MSSQLSERVER¥test¥pngtable¥sample5.png	84F915B4-F4D...	0x89504E470D0A1A0A00000
4	sample6.png	¥¥WIN10¥MSSQLSERVER¥test¥pngtable¥sample6.png	86F915B4-F4D...	0x89504E470D0A1A0A00000
5	sample7.png	¥¥WIN10¥MSSQLSERVER¥test¥pngtable¥sample7.png	88F915B4-F4D...	0x89504E470D0A1A0A00000
6	sample8.png	¥¥WIN10¥MSSQLSERVER¥test¥pngtable¥sample8.png	8AF915B4-F4D...	0x89504E470D0A1A0A00000
7	sample9.png	¥¥WIN10¥MSSQLSERVER¥test¥pngt...¥sample9.png	8CF915B4-F4...	0x89504E470D0A1A0A00000
8	sample0.png	¥¥WIN10¥MSSQLSERVER¥test¥p...	...F4D...	0x89504E470D0A1A0A00000
9	sample1.png	¥¥WIN10¥MSSQLSERVER¥test¥p...	...F4D...	0x89504E470D0A1A0A00000
10	sample2.png	¥¥WIN10¥MSSQLSERVER¥test¥p...	...F4D...	0x89504E470D0A1A0A00000

この手書きの画像をもとに、CNN で作成したモデルで数字の予測を行うには、次のように **sp_execute_external_script** ストアド プロシージャを実行します。

```
-- FileTable からデータを取得して、予測を実行
USE fsTestDB

EXEC sp_execute_external_script
 @language = N'Python'
,@script = N'
from cntk.ops.functions import load_model
z = load_model("c:/temp/model.sav")

import numpy as np
import scipy.misc
file_path = InputDataSet["filepath"]
img_array = scipy.misc.imread(file_path[0].replace("¥¥","/"), flatten=True)
img_array = scipy.misc.imresize(img_array, (28,28))   # 28*28 にサイズ変更
img_data = 255.0 - img_array.reshape(784)             # 白黒反転
img_data = img_data.reshape(1,28,28)

predictions = np.squeeze(z.eval({z.arguments[0]:[img_data]}))
print(predictions)
print(np.argmax(predictions))
'
,@input_data_1 = N'SELECT FileTableRootPath()
                        + file_stream.GetFileNamespacePath() AS filepath
                  FROM fTable1 WHERE name = ''sample8.png'' '
```

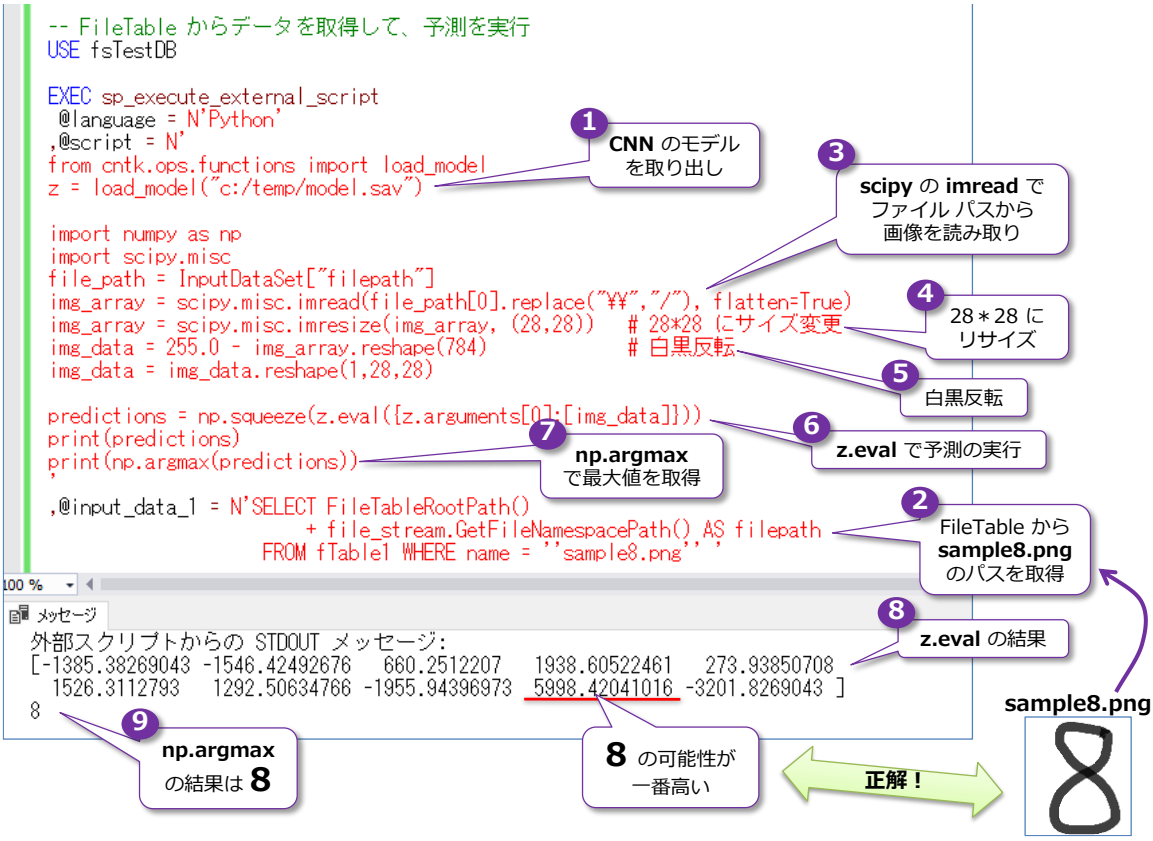
最初に「**load_model("c:/temp/model.sav")**」でファイルとして保存した CNN のモデルを取り出して（ロードして）、次に、**@input_data_1** で指定した SELECT ステートメントで FileTable 内の「**sample8.png**」という名前のファイルのファイル パスをもとに、「**scipy**」ライブラリの「**imread**」メソッドで画像を読み込んでいます。指定するファイル名は、皆さんの環境に合わせて適宜変更してください。imread でファイルを読み取るときに、ファイル パス内の「**¥**」

マークを「**/**」（スラッシュ）に置換（replace メソッドで全置換）しています。

次の「**imresize**」メソッドでは 28*28 の大きさにサイズを変更しています（MNIST の 1 つ 1 つのデータは、28*28 の大きさなので、このサイズに合わせています）。

次の「**255.0 - ～**」では、画像の白黒反転を行っています（MNIST のデータは、背景色が黒で、数字が白のため、白が背景の手書き数字の場合は、白黒反転が必要になります）。

次の「**z.eval**」が実際に予測行っている部分です。この結果（**predictions**）は、「**[-1385.38269043 -1546.42492676 660.2512207 …]**」のように数字が 10 個返ってきます。これは、**[0 1 2 3 4 5 6 7 8 9]** という順番で、0～9 までの数字で、どの数字の可能性が一番高いのか（予測の結果）が返ってきます。数字が一番大きいものが、その数字の可能性が高いことになるので、次の「**np.argmax**」（numpy の argmax メソッド）で一番大きい値を取り出しています。



以上のように、Microsoft Cognitive Toolkit を利用すれば、簡単に画像認識モデルを作成することができます。前掲のチュートリアル サイトは本当に充実しているので、これを参考にいろいろなモデルを試してみていただければと思います。

STEP 4. R による機械学習やその他の利用方法

この STEP では、R を利用したクラスタリング（k-means 法）の利用方法や、ML Services が利用するメモリ使用量の調整方法、モデル作成時のデータの取得件数を制御する方法など、その他の利用方法を説明します。

この STEP では、次のことを学習します。

- ✓ R を利用したクラスタリング（k-means 法）
- ✓ ML Services が利用するメモリ使用量の調整（リソース ガバナー）
- ✓ モデル作成時のデータの取得件数を制御（rowsPerRead）

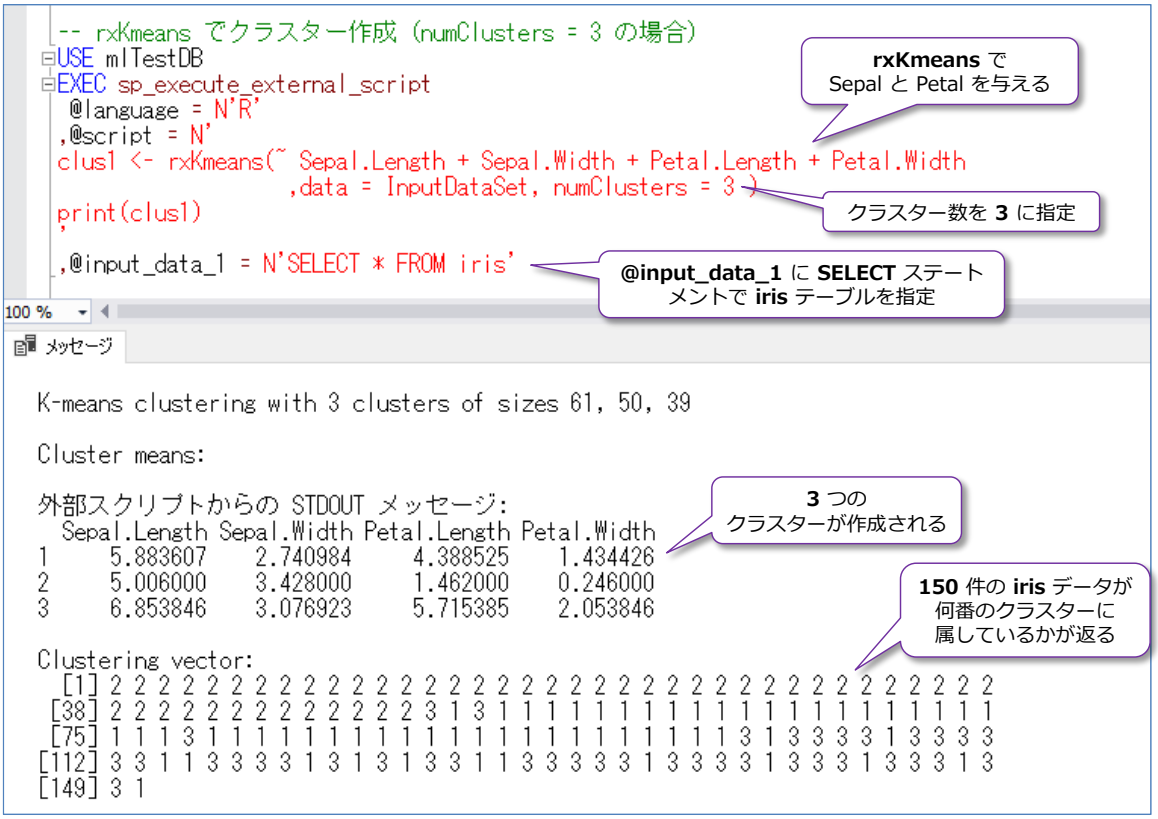
4.1 R を利用したクラスタリング（k-means 法）

まずは、R でのクラスタリングの利用方法について説明します。クラスタリングは、顧客のセグメンテーションやデータを分類する際に、よく利用される機械学習の 1 つです。クラスタリングで定番となっているのは、「**k-means**」（k 平均法）というアルゴリズムですが、RevoScaleR では「**rxKmeans**」関数で利用することができます。

Let's Try

それでは、これも試してみましょう。

```
USE mlTestDB
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
clus1 <- rxKmeans(~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                  ,data = InputDataSet, numClusters = 3 )
print(clus1)
'
,@input_data_1 = N'SELECT * FROM iris'
```

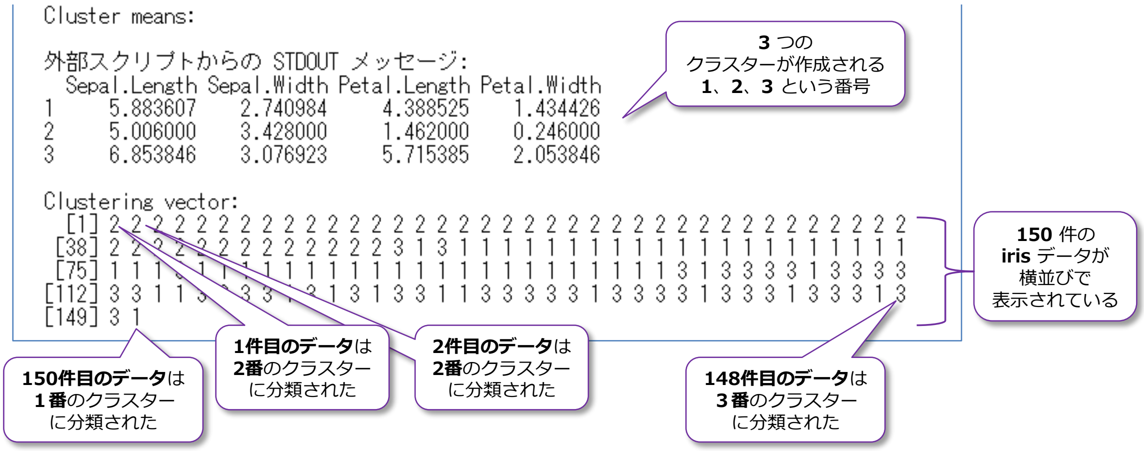


これまで利用してきた機械学習のアルゴリズムでは、「**目的変数 ~ 説明変数１ + 説明変数２ + 説明変数３ + …**」という形で、目的変数に **Species**（アヤメの種類）を与えて、モデルを作成しましたが、rxKmeans では、目的変数は不要で「**~ 説明変数１ + 説明変数２ + 説明変数３ + …**」

という形で利用します（チルダを先頭で指定します）。

なお、これまで利用してきた機械学習のアルゴリズム（決定木やランダム フォレスト、サポート ベクター マシン、ニューラル ネットワークなど）は、「**教師あり学習**」（Supervised learning）とも呼ばれていて、正解データ（正解ラベルとも呼ばれます）を目的変数として与えてモデルを作成するというものでした。これに対して、クラスタリングは「**教師なし学習**」とも呼ばれていて、説明変数をもとに、データ分類を実施するというものになっています（正解データは不要です）。

「**print(clus1)**」で出力したクラスターの結果は、次のように解釈します。



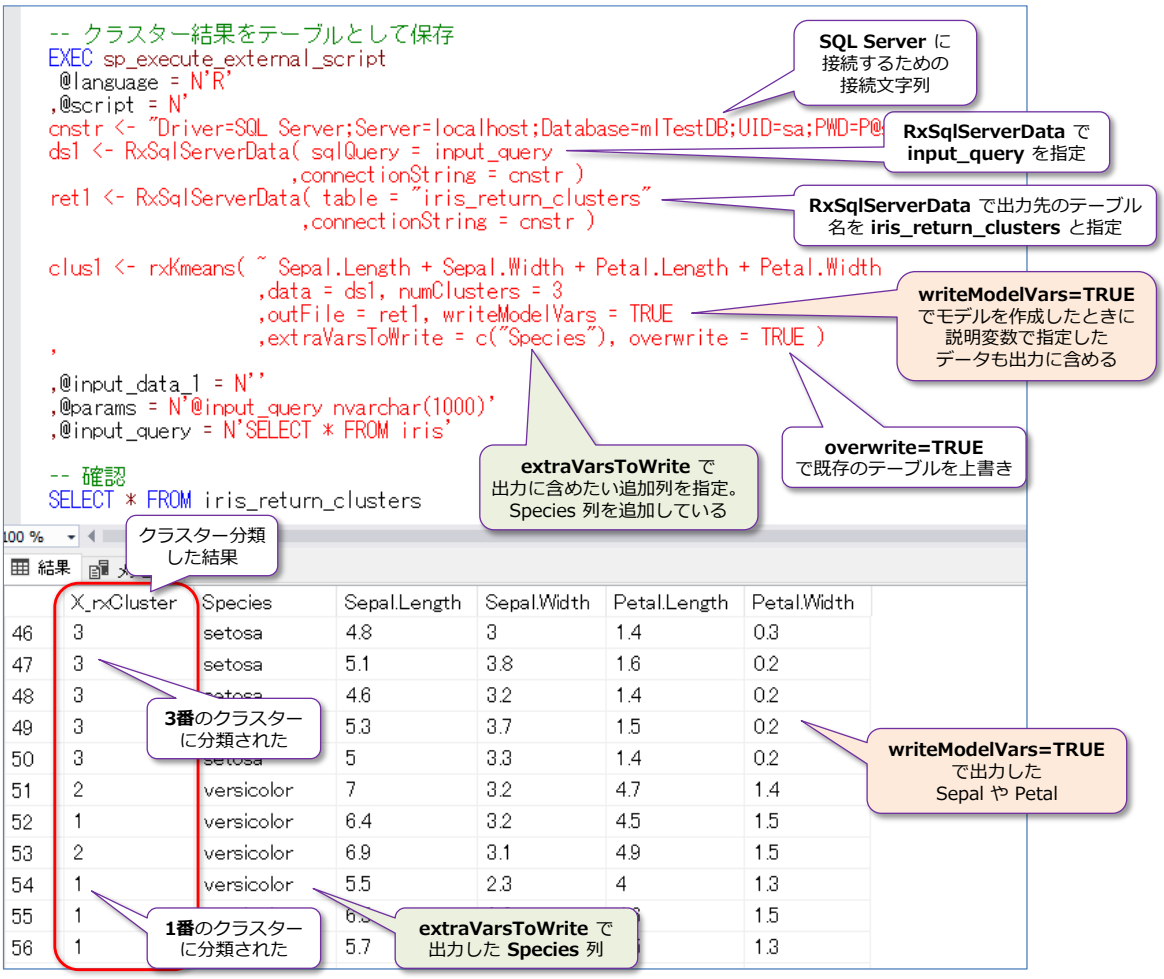
150 件のデータが横並びで表示されて、何番のクラスターに属しているのかを確認できます。

rxKmeans の結果をテーブルに保存する

次に、**rxKmeans** の結果を SQL Server 上のテーブルとして保存してみましょう。これを行うには、RevoScaleR の「**RxSqlServerData**」関数を利用します（以下のように記述します）。

```
-- クラスター結果をテーブルとして保存
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
cnstr <- "Driver=SQL Server;Server=localhost;Database=mlTestDB;UID=sa;PWD=saﾊﾟｽﾜｰﾄﾞ;"
ds1 <- RxSqlServerData( sqlQuery = input_query
                       ,connectionString = cnstr )
ret1 <- RxSqlServerData( table = "iris_return_clusters"
                        ,connectionString = cnstr )

clus1 <- rxKmeans( ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                  ,data = ds1, numClusters = 3
                  ,outFile = ret1, writeModelVars = TRUE
                  ,extraVarsToWrite = c("Species"), overwrite = TRUE )
'
,@input_data_1 = N''
,@params = N'@input_query nvarchar(1000)'
,@input_query = N'SELECT * FROM iris'
```

```
-- 結果の確認（iris_return_clusters という名前のテーブルが自動作成されている）
SELECT * FROM iris_return_clusters
```



RxSqlServerData 関数では、接続文字列を利用して、SQL Server に接続するので、最初に「**cnstr <-**」という形で Server= や Database= などで接続先の SQL Server 名とデータベース名を指定しています。

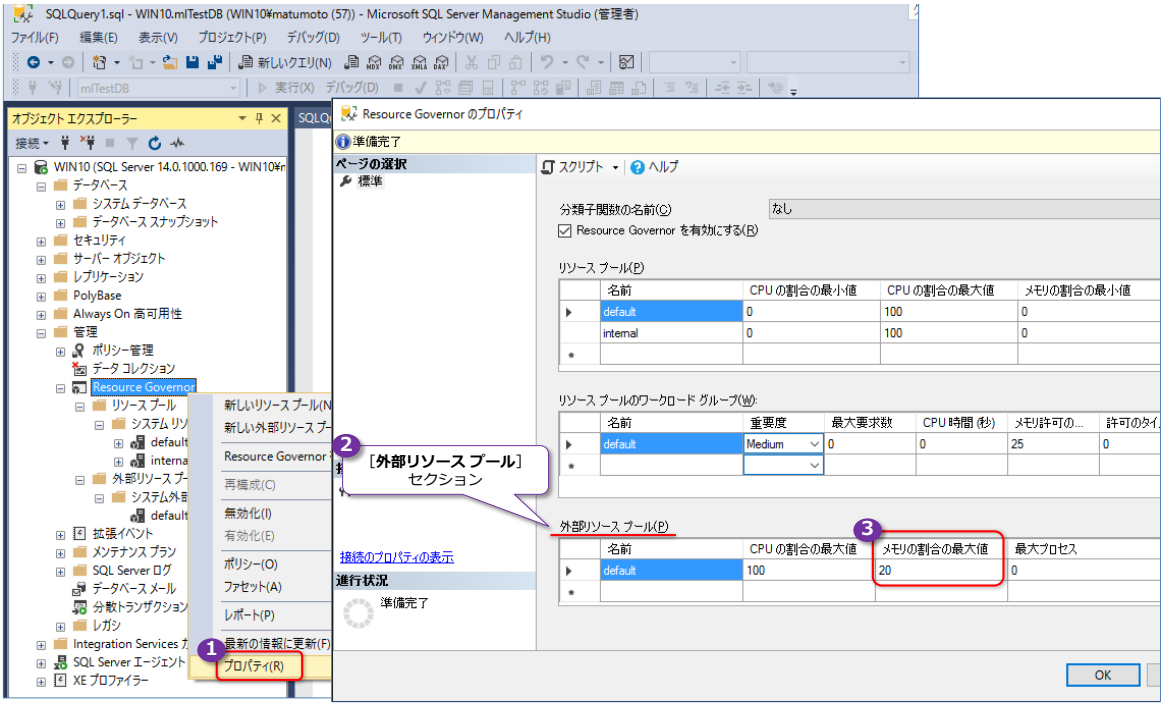
次の「**ds1 <-**」では「**@input_query**」入力変数で指定した **SELECT** ステートメント（iris テーブル）を「**sqlQuery = input_query**」という形で指定します。**RxSqlServerData** 関数では、今まで利用してきた「**@input_data_1**」パラメーターではなく、別途「**@params**」で入力変数を定義して、そこに SELECT ステートメントを記述する必要があります。

次の「**ret1 <-**」では、出力結果のテーブル名を定義していますが、「**table="iris_return_clusters"**」とすることで、iris_return_clusters という名前のテーブルを SQL Server 上に自動作成することができます（後述の **rxKmeans** の引数で **outFile = ret1** と指定することで、**rxKmeans** の出力結果をテーブルとして自動作成できます）。

rxKmeans の引数では、**writeModelVars=TRUE** を指定することで説明変数（Sepal や Petal）、**extraVarsToWrite** で **Species** 列を指定することで、出力結果を分かりやすくすることができます。もし、これらを付けない場合は、クラスター番号だけが 150 件分出力される形になります。

4.2 ML Services が利用するメモリ使用量の調整（リソース ガバナー）

SQL Server 2017 の Machine Learning Services が利用するメモリ使用量は、既定で **20%** に設定されています。これは、リソース ガバナーで調整されていて、次のように、オブジェクト エクスプローラーで［**管理**］フォルダーを展開して、［**Resource Governor**］から確認することができます。



［**Resource Governor**］を右クリックして、［**プロパティ**］を開き、［**外部リソース プール**］セクションの「**メモリの割合の最大値**」が Machine Learning Services が利用できるメモリの最大量になっています。既定の 20% の場合は、大量のデータを扱う場合にメモリ不足になる可能性があるので、メモリ不足で実行エラーになる場合は、この値を修正してみてください。

なお、Transact-SQL ステートメントで設定を確認する場合は、次のように「**resource_governor_external_resource_pools**」ビューを参照します。

```
-- 外部リソースプールの確認。20% に設定されていることの確認
SELECT * FROM sys.resource_governor_external_resource_pools
```

```
-- 外部リソースプールの確認。20% に設定されていることの確認
SELECT * FROM sys.resource_governor_external_resource_pools
```

100 %

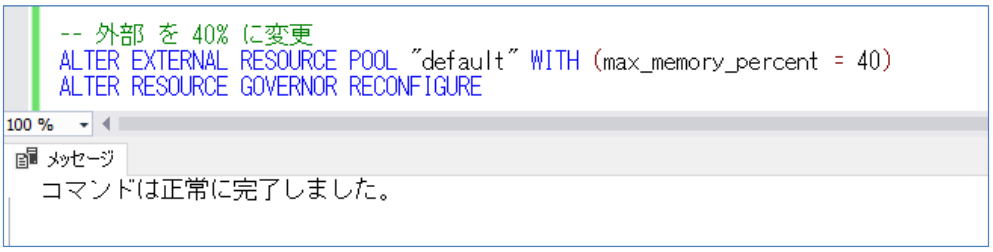
結果 メッセージ

	external_pool_id	name	max_cpu_percent	max_memory_percent	max_processes	version
1	2	default	100	20	0	4

max_memory_percent が、前掲の「**メモリの割合の最大値**」に該当する場所になっています。この値を Transact-SQL ステートメントで変更したい場合は、次のように **ALTER EXTERNAL**

RESOURCE POOL ステートメントを実行します。

```
-- 外部リソース プールを 40% に変更
ALTER EXTERNAL RESOURCE POOL "default" WITH (max_memory_percent = 40)
ALTER RESOURCE GOVERNOR RECONFIGURE
```



「**default**」という名前のリソース プールを利用しているので、これに対して **max_memory_percent** の値を指定することで、メモリ使用量の割合を変更することができます。

メモリ使用量は、ユーザー定義のリソース プールを作成しても行うことができます。これについては、オンライン ブック（SQL Server のヘルプ）の以下のトピックが参考になります。

機械学習用リソース プールを作成します。
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/r/how-to-create-a-resource-pool-for-r

SQL Server の Machine Learning のサービスの Dmv
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/r/dmvs-for-sql-server-r-services

4.3 モデル作成時のデータ処理件数を制御（rowsPerRead）

RevoScaleR では、機械学習のモデルを作成する際に、データの処理件数を制御することができます。これは、大量データの際に、メモリ使用量を抑えるための実装であり、実行時間（モデルの作成時間）とメモリ使用量のトレードオフになります。

データの処理件数を制御するには、次のように **RxSqlServerData** 関数を利用します。

```
-- rowsPerRead でデータの処理件数を制御
EXEC sp_execute_external_script
 @language = N'R'
,@script = N'
cnstr <- "Driver=SQL Server;Server=localhost;Database=mlTestDB;UID=sa;PWD=saﾊﾟｽﾜｰﾄﾞ;"
ds1 <- RxSqlServerData( sqlQuery = input_query
                       ,connectionString = cnstr
                       ,stringsAsFactors = TRUE
                       ,rowsPerRead = 1000000 )

model1 <- rxDForest(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
                    ,data = ds1, nTree = 10)
print(model1)
'
,@input_data_1 = N''
,@params = N'@input_query nvarchar(1000)'
,@input_query = N'SELECT * FROM iris'
```

RxSqlServerData 関数では、接続文字列を利用して、SQL Server に接続するので、最初に「**cnstr <-**」で、Server= や Database= などw利用して接続先の SQL Server 名とデータベース名を指定しています。

次の「**ds1 <-**」では「**@input_query**」入力変数で指定した **SELECT** ステートメント（iris テーブル）を「**sqlQuery = input_query**」という形で指定します。

ここで、「**rowsPerRead = 1000000**」と記述することで、モデル作成時のデータの処理件数を 100 万件ずつに設定することができます。今回の iris データは 150 件のデータしかないので、この 100 万という指定は意味がないものですが、このように **rowsPerRead** を指定することで、大量データを対象とする場合には、少しずつデータを処理して、利用するメモリ量を抑えられるようになります（もし、メモリ不足で実行エラーになる場合は、この **rowsPerRead** を小さくしてみてください）。

RxSqlServerData では「**stringsAsFactors = TRUE**」も指定していますが、R での **factor** は目的変数に指定したもの（この例では **Species**）で、Species 列のデータは setosa や virginica などの文字列データ（string）なので、**TRUE** に設定することで、モデルの作成ができるようになります（既定値は FALSE なので、文字列の場合にはエラーになります）。

あとは、モデルを作成するときに、「**data = ds1**」と記述して、**RxSqlServerData** で作成したデータ ソースを指定すれば、**rowsPerRead** の設定が有効になります。

4.4 その他の参考資料

SQL Server 2017 の Machine Learning Services を利用するにあたっては、オンライン ブック（SQL Server のヘルプ）の以下のトピックが参考になります。

SQL Server 2017 Machine Learning Services のヘルプのトップページ
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/getting-started-with-machine-learning-services

SQL Server の machine learning のチュートリアル
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/tutorials/machine-learning-services-tutorials

Get Started with SQL Server Machine Learning Services
https://microsoft.github.io/sql-ml-tutorials/

Machine Learning のサービスの高度な構成オプション
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/r/configure-and-manage-advanced-analytics-extensions

事前トレーニング済みの機械学習の SQL Server 上のモデルをインストールします。
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/r/install-pretrained-models-sql-server

SQL Server Team Blog: n-database Machine Learning in SQL Server 2017
https://blogs.technet.microsoft.com/dataplatforminsider/2017/09/26/in-database-machine-learning-in-sql-server-2017/

Use Python with revoscalepy to create a model
https://docs.microsoft.com/en-us/sql/advanced-analytics/tutorials/use-python-revoscalepy-to-create-model

おわりに

最後まで試された皆さん、いかがでしたでしょうか？ SQL Server 2017 の Machine Learning Services は、いろいろな可能性を持った機能であることを確認できたのではないでしょうか。この自習書では iris データを利用しましたが、皆さんの実際のデータでも、ぜひ機械学習を試してみていただければと思います（現在、Machine Learning Services は日本語（ダブルバイト）の列名には対応していないので、実際のデータに日本語がある場合は、SELECT ステートメントの実行時に AS で英語／半角の別名を付けて実行したり、英語の列名を付けたビューを作成するなどして、実行してみてください）。

SQL Server 2017 には、Linux 対応や自動チューニング機能の搭載、グラフ データベース、クエリ ストアの強化（クエリの Wait 情報を過去に遡って参照可能）、列ストア インデックス／インメモリ OLTP のさらなる進化、クラスター レス可用性グループ、スマート バックアップなど、役立つ機能が盛りだくさんで提供されています。

また、BI 機能も進化しており、Analysis Services に Power BI（データ取得部分での Power Query）の統合、Reporting Services のレポート コメント、Integration Services のスケールアウト実行など、現場で役立つ機能が提供されています。

こうした SQL Server 2017 の新機能については、本自習書シリーズの「**No.1 SQL Server 2017 の新機能の概要**」編および「**No.2 SQL Server 2017 on Linux の概要**」編で詳しく説明しているので、こちらもぜひご覧いただければと思います。

執筆者プロフィール

有限会社エスキューエル・クオリティ（http://www.sqlquality.com/）

SQLQuality（エスキューエル・クオリティ）は、**日本で唯一の SQL Server 専門の独立系コンサルティング会社**です。過去のバージョンから最新バージョンまでの SQL Server を知りつくし、多数の実績と豊富な経験を持つ、OS や .NET にも詳しい **SQL Server の専門家（キャリア 20 年以上）がすべての案件に対応します**。人気メニューの「**パフォーマンス チューニング サービス**」は、100%の成果を上げ、過去すべてのお客様環境で驚異的な性能向上を実現。チューニング スキルは**世界トップレベル**を自負、検索エンジンでは（英語情報を含めて）ヒットしないノウハウを多数保持。ここ数年は **BI／DWH システム構築支援**のご依頼が多く、支援だけでなく実際の構築も行う。

主なコンサルティング実績／構築実績

- 大手製造業の「**CAD 端末の利用状況の見える化**」システム構築
  Oracle や CSV（Notes）、TSV ファイル、Excel からデータを抽出し、SQL Server 2012 上に DWH を構築
  見える化レポートには Reporting Services を利用
- 大手映像制作会社の **BI システム構築**（会計／業務システムにおける予実管理／原価管理など）
  従来 Excel で管理していたシートを Reporting Services のレポートへ完全移行。
  Oracle や勘定奉行からデータを抽出して、SQL Server 上に DWH を構築
- 大手流通系の **DWH／BI システム構築支援**（POS データ／在庫データ分析／ABC 分析／ポイントカード分析）
- 大手アミューズメント企業の **BI システム構築支援**（人事システムにおける人材パフォーマンス管理）
  Reporting Services による勤怠状況の見える化レポートの作成、PostgreSQL／人事システムからのデータ抽出
- 外資系医療メーカーの **BI システム**構築支援（Analysis Services と Excel による販売分析システム）
  OLAP キューブによる売上および顧客データの多次元分析／自由分析（ユーザーによる自由操作が可能）
- 大手流通系の **DWH システム**のパフォーマンス チューニング
  データ量 100 億件の DWH、**総ステップ数 2 万**越えのストアド プロシージャのパフォーマンス チューニング
- ミッション クリティカルな**金融システム**でのトラブル シューティング／定期メンテナンス支援
- SQL Server の下位バージョンからの**移行／アップグレード**支援（32 ビットから x64 への対応も含む）
- 複数台の SQL Server の **Hyper-V 仮想環境**への移行支援（サーバー統合支援）
- ハードウェア リプレース時の**ハードウェア選定**（最適なサーバー、ストレージの選定）、**高可用性環境**の構築
- **2 時間**かかっていた日中バッチ実行時間を、わずか **5 分**へ短縮（**95.8%** の性能向上）
- **Java 環境**（Tomcat、Seasar2、S2Dao）の SQL Server パフォーマンス チューニング etc

コンサルティング時の作業例（パフォーマンス チューニングの場合）

- アプリケーション コード（VB、C#、Java、ASP、VBScript、VBA）の解析／改修支援
- ストアド プロシージャ／ユーザー定義関数／トリガー（Transact-SQL）の解析／改修支援
- インデックス チューニング／SQL チューニング／ロック処理の見直し
- 現状のハードウェアで将来のアクセス増にどこまで耐えられるかを測定する高負荷テストの実施
- IIS ログの解析／アプリケーション ログ（log4net／log4j）の解析
- ボトルネック ハードウェアの発見／ボトルネック SQL の発見／ボトルネック アプリケーションの発見
- SQL Server の構成オプション／データベース設定の分析／使用状況（CPU, メモリ, ディスク, Wait）解析
- 定期メンテナンス支援（インデックスの再構築／断片化解消のタイミングや断片化の事前防止策など）etc

松本美穂（まつもと・みほ）

有限会社エスキューエル・クオリティ 代表取締役 Microsoft MVP for SQL Server（2004 年 4 月～）
経産省認定データベース スペシャリスト／MCDBA／MCSD for .NET／MCITP Database Administrator
SQL Server の日本における最初のバージョンである「SQL Server 4.21a」から SQL Server に携わり、現在、SQL Server を中心とするコンサルティングを行っている。得意分野はパフォーマンス チューニングと Reporting Services。著書の『SQL Server 2000 でいってみよう』と『ASP.NET でいってみよう』（いずれも翔泳社刊）は、トップ セラー（前者は 28,500 部、後者は 16,500 部発行）。近刊に『SQL Server 2016 の教科書』（ソシム刊）がある。

松本崇博（まつもと・たかひろ）

有限会社エスキューエル・クオリティ 取締役 Microsoft MVP for SQL Server（2004 年 4 月～）
経産省認定データベース スペシャリスト／MCDBA／MCSD for .NET／MCITP Database Administrator
SQL Server の BI システムとパフォーマンス チューニングを得意とするコンサルタント。アプリケーション開発（ASP／ASP.NET、C#、VB 6.0、Java、Access VBA など）やシステム管理者（IT Pro）経験もあり、SQL Server だけでなく、アプリケーションや OS、Web サーバーを絡めた、総合的なコンサルティングが行えるのが強み。Analysis Services と Excel による BI システムも得意とする。マイクロソフト認定トレーナー時代の 1998 年度には、Microsoft CPLS トレーナー アワード（Trainer of the Year）を受賞。