**Универсальный подход (почти) к любой задаче машинного обучения**

03.08.2016

Теги: [машинное обучение](http://datareview.info/tag/mashinnoe-obuchenie/)

Любой data scientist ежедневно работает с большими объемами данных. Считается, что около 60% – 70% времени занимает первый этап рабочего процесса: очистка, фильтрация и преобразование данных в формат, подходящий для применения алгоритмов машинного обучения. На втором этапе выполняется предобработка и непосредственное обучение моделей. В сегодняшней статье мы сконцентрируемся на втором этапе процесса и рассмотрим различные техники и рекомендации, являющиеся результатом моего участия более чем в 100 соревнованиях по машинному обучению. Несмотря на то, что описанные концепции имеют достаточно общий характер, они будут полезны при решении многих конкретных задач.

Все примеры кода написаны на Python!

Данные

Перед тем, как применять алгоритмы машинного обучения, данные необходимо преобразовать в табличное представление. Этот процесс, представленный на рисунке ниже, является наиболее сложным и трудоемким.

*Data in Database – данные в базе данных. Data Munging – фильтрация данных. Useful Data – полезные данные. Data Conversion – преобразование данных. Tabular Data – табличные данные. Data – независимые переменные (признаки). Labels – зависимые переменные (целевые переменные).*

После того, как данные преобразованы в табличное представление, их можно использовать для обучения моделей. Табличное представление является наиболее распространенным представлением данных в сфере машинного обучения и интеллектуального анализа данных. Строки таблицы являются отдельными объектами (наблюдениями). Столбцы таблицы, содержат независимые переменные (признаки), обозначаемые *X*, и зависимые (целевые) переменные, обозначаемые *y*. В зависимости от класса задачи, целевые переменные могут быть представлены как одним, так и несколькими столбцами.

Типы целевых переменных

Целевые переменные определяют класс задачи и могут быть представлены одним из следующих вариантов:

* Один столбец с двоичными значениями: задача двухклассовой классификации (binary classification), каждый объект принадлежит только одному классу.
* Один столбец с действительными значениями: задача регрессии, прогнозируется одна величина.
* Несколько столбцов с двоичными значениями: задача многоклассовой классификации (multi-class classification), каждый объект принадлежит только одному классу.
* Несколько столбцов с действительными значениями: задача регрессии, прогнозируется несколько величин.
* Несколько столбцов с двоичными значениями: задача классификации с пересекающимися классами (multi-label classification), один объект может принадлежать нескольким классам.

Метрика

При решении любой задачи машинного обучения, необходимо иметь возможность оценивать результат, то есть, необходима метрика. Например, для задачи двухклассовой классификации в качестве метрики обычно используется площадь под ROC-кривой (ROC AUC). В случае многоклассовой классификации обычно применяется категорийная кросс-энтропия (categorical cross-entropy). В случае регрессии – среднее квадратов отклонений (mean squared error, MSE).

Мы не будем подробно рассматривать метрики, поскольку они могут быть достаточно разнообразны и выбираются под конкретную задачу.

Библиотеки

Первым делом необходимо установить базовые библиотеки, необходимые для выполнения вычислений, такие как *numpy* и *scipy*. Затем можно приступить к установке наиболее популярных библиотек для анализа данных и машинного обучения:

* Исследование и преобразование данных: *pandas* (<http://pandas.pydata.org/>).
* Широкий спектр различных алгоритмов машинного обучения: *scikit-learn* (<http://scikit-learn.org/stable/>).
* Лучшая реализация градиентного бустинга (gradient boosting): *xgboost* (<https://github.com/dmlc/xgboost>).
* Нейронные сети: *keras* (<http://keras.io/>).
* Визуализация: *matplotlib* (<http://matplotlib.org/>).
* Индикатор прогресса выполнения: *tqdm* (<https://pypi.python.org/pypi/tqdm>).

Следует сказать, что я не использую *Anaconda* (<https://www.continuum.io/downloads>). *Anaconda* объединяет в себе большинство популярных библиотек и существенно упрощает процесс установки, но мне необходимо больше свободы. Решать вам.

Фреймворк для машинного обучения

В 2015 году я представил концепцию фреймворка для автоматического машинного обучения. Система еще находится в стадии разработки, но скоро будет сделан релиз. Структура фреймворка, которая послужит основой для дальнейшего изложения, представлена на рисунке ниже.

Рисунок из публикации: Такур А., Крон-Гримберге А. AutoCompete: фреймворк для соревнований по машинному обучению. ([A. Thakur and A. Krohn-Grimberghe, AutoCompete: A Framework for Machine Learning Competitions.](https://arxiv.org/abs/1507.02188))

На входе фреймворк получает данные, предварительно преобразованные в табличное представление. Розовые линии показывают направление для простейшего случая.

На первом шаге определяется класс задачи. Это можно сделать, проанализировав целевую переменную. Задача может представлять собой классификацию или регрессию, классификация может быть двухклассовой или многоклассовой, классы могут пересекаться или не пересекаться. После того, как класс задачи определен, мы разделяем исходный набор данных на две части и получаем обучающий набор (training set) и валидационный набор (validation set), как показано на рисунке ниже.

В том случае, если мы имеем дело с классификацией, разделение данных необходимо выполнить таким образом, чтобы в полученных наборах соотношение количеств объектов, относящихся к разным классам, соответствовало этому соотношению для исходного набора данных (stratified splitting). Это легко можно сделать с помощью класса *StratifiedKFold* библиотеки *scikit-learn*.

Для задачи регрессии подойдет обычное разделение с помощью класса *KFold*, который также доступен в библиотеке *scikit—learn*.

Кроме того, для задачи регрессии существуют более сложные методы разделения данных, обеспечивающие одинаковое распределение целевой переменной в полученных наборах. Эти подходы остаются на самостоятельное рассмотрение читателю.

В примере кода выше размер валидационного набора (*eval\_size*) составляет 10% от исходного набора данных. Это значение следует выбирать, ориентируясь на объем исходных данных.

После разделения данных все операции, применяемые к обучающему набору, необходимо сохранять, а затем применять к валидационному набору. Валидационный набор ни в коем случае нельзя объединять с обучающим набором. Если это сделать, мы будем получать очень хорошие оценки, при этом наши модели будут бесполезны в виду сильного переобучения.

На следующем шаге мы определяем типы признаков. Наиболее распространены три типа: числовой, категорийный и текстовый. Давайте рассмотрим набор данных из популярной задачи о пассажирах «Титаника» (<https://www.kaggle.com/c/titanic/data>).

|  |  |
| --- | --- |
| Описание переменных | |
| *survival* | Выжил ли пассажир (0 – нет, 1 – да) |
| *pclass* | Класс (1 – первый, 2 – второй, 3 – третий) |
| *name* | Имя |
| *sex* | Пол |
| *age* | Возраст |
| *sibsp* | Количество братьев, сестер, супругов на борту |
| *parch* | Количество родителей, детей на борту |
| *ticket* | Номер билета |
| *fare* | Стоимость билета |
| *cabin* | Каюта |
| *embarked* | Место посадки (C – Шербур, Q – Куинстаун, S – Саутгемптон) |

В этом наборе данных столбец *survival* содержит целевую переменную. На предыдущем шаге мы уже отделили целевую переменную от признаков. Признаки *pclass*, *sex* и *embarked* являются категорийными. Признаки *age*, *sibsp*, *parch* и подобные являются числовыми. Признак *name* является текстовым. Впрочем, я не думаю, что имя пассажира будет полезно при прогнозировании, выжил этот пассажир или нет.

Числовые признаки не нуждаются в преобразовании. Такие признаки в исходном виде готовы к нормализации и обучению моделей.

Категорийные признаки можно преобразовать двумя способами:

* Присвоить каждой уникальной категории уникальное целое число. Это можно сделать с помощью класса *LabelEncoder* библиотеки *scikit—learn*.
* Результат предыдущего преобразования далее можно преобразовать в двоичные признаки, то есть в унитарный код (one-hot encoding). Это можно сделать с помощью класса *OneHotEncoder* библиотеки *scikit—learn*.

[[Рисунок]](https://media.licdn.com/mpr/mpr/shrinknp_800_800/AAEAAQAAAAAAAAiTAAAAJDQ4NmQ5NzM4LTMyOTctNDBlNS1iYWQ3LWU3NGUzZDE5ODQ5Ng.png)

Не забудьте, что перед применением *OneHotEncoder*, необходимо преобразовать категории в числа с помощью *LabelEncoder*.

Поскольку данные из соревнования «Титаник» не содержат хорошего примера текстового признака, давайте сформулируем общее правило преобразование текстовых признаков. Мы можем объединить все текстовые признаки в один, а затем применить соответствующие алгоритмы, позволяющие преобразовать текст в числовое представление.

Текстовые признаки можно объединить следующим образом:

Затем мы можем выполнить преобразование с помощью класса *CountVectorizer* или *TfidfVectorizer*библиотеки *scikit-learn*.

Как правило, *TfidfVectorizer* обеспечивает лучший результат, чем *CountVectorizer*. На практике я выяснил, что следующие значения параметров *TfidfVectorizer* являются оптимальными в большинстве случаев:

Если вы применяете векторизатор только к обучающему набору, не забудьте сохранить его на диске, чтобы в последствии применить к валидационному набору.

На следующем шаге признаки, полученные в результате описанных выше преобразований, передаются в стекер (stacker). Этот узел фреймворка объединяет все преобразованные признаки в одну матрицу. Обратите внимание, в нашем случае речь идет о стекере признаков (feature stacker), который не следует путать со стекером моделей (model stacker), представляющим другую популярную технологию.

Объединение признаков можно выполнить с помощью функции *hstack* библиотеки *numpy* (в случае неразреженных (dense) признаков) или с помощью функции *hstack* из модуля *sparse* библиотеки *scipy* (в случае разреженных (sparse) признаков).

В том случае, если выполняются другие этапы предобработки, например, уменьшение размерности или отбор признаков (будут рассмотрены далее), объединение полученных признаков можно эффективно выполнить с помощью класса  *FeatureUnion* библиотеки *scikit—learn*.

После того, как все признаки объединены в одну матрицу, мы можем приступать к обучению моделей. Поскольку признаки не нормализованы, на данном этапе следует применять только ансамблевые алгоритмы на основе деревьев решений:

* *RandomForestClassifier*
* *RandomForestRegressor*
* *ExtraTreesClassifier*
* *ExtraTreesRegressor*
* *XGBClassifier*
* *XGBRegressor*

Чтобы применять линейные модели, необходимо выполнить нормализацию признаков с помощью классов *Normalizer* или *StandardScaler* библиотеки *scikit-learn*.

Данные методы нормализации обеспечивают хороший результат только в случае неразреженных (dense) признаков. Чтобы применить *StandardScaler* к разреженным (sparse) признакам, в качестве параметра необходимо указать *with\_mean=False*.

Если описанные выше шаги обеспечили для нас «хорошую» модель, можно переходить к настройке гиперпараметров. Если же модель нас не удовлетворяет, мы можем продолжить работу с признаками. В частности, в качестве дополнительных шагов мы можем применить различные методы уменьшения размерности.

Чтобы не усложнять, мы не будем рассматривать линейный дискриминантный анализ (linear discriminant analysis, LDA) и квадратичный дискриминантный анализ (quadratic discriminant analysis, QDA). В общем случае, для уменьшения размерности данных применяется метод главных компонент (principal component analysis, PCA). При работе с изображениями следует начинать с 10 – 15 компонент и увеличивать это значение до тех пор, пока результат существенно улучшается. При работе с другими видами данных можно начинать с 50 – 60 компонент.

В случае текстовых данных, после преобразования текста в разреженную матрицу можно применить сингулярное разложение (singular value decomposition, SVD). Реализация сингулярного разложения *TruncatedSVD* доступна в библиотеке *scikit-learn*.

Количество компонент сингулярного разложения, которое, как правило, обеспечивает хороший результат в случае признаков, полученных в результате преобразования с помощью *CountVectorizer* или *TfidfVectorizer*, составляет 120 – 200. Большее количество компонент позволяет незначительно улучшить результат ценой существенных вычислительных затрат.

Выполнив описанные шаги, не забываем нормализовать признаки, чтобы иметь возможность применять линейные модели. Далее мы можем либо использовать подготовленные признаки для обучения моделей, либо выполнить отбор признаков (feature selection).

Существуют различные методы отбора признаков. Одним из популярных методов является жадный алгоритм отбора признаков (greedy feature selection). Жадный алгоритм имеет описанную далее схему. Шаг 1: обучаем и оцениваем модель на каждом из исходных признаков; отбираем один признак, обеспечивший лучшую оценку. Шаг 2: обучаем и оцениваем модель на парах признаков, состоящих из лучшего признака, отобранного на предыдущем шаге, и каждого из оставшихся признаков; отбираем лучший признак из оставшихся. Повторяем аналогичные шаги, пока не отберем нужное количество признаков, или пока не будет выполнен какой-либо другой критерий. Вариант реализации данного алгоритма, где в качестве метрики используется площадь под ROC-кривой, доступен по следующей ссылке: <https://github.com/abhishekkrthakur/greedyFeatureSelection>. Следует отметить, что данная реализация неидеальна и требует определенных модификаций под конкретную задачу.

Другим более быстрым методом отбора признаков является отбор с помощью одного из алгоритмов машинного обучения, которые оценивают важность признаков. Например, можно использовать логистическую регрессию (logistic regression) или случайный лес (random forest). В дальнейшем отобранные признаки можно использовать для обучения других алгоритмов.

Выполняя отбор признаков с помощью случайного леса, помните, что количество деревьев должно быть небольшим, кроме того, не следует выполнять серьезную настройку гиперпараметров, иначе возможно переобучение.

Отбор признаков также можно выполнить с помощью алгоритмов на основе градиентного бустинга. Рекомендуется использовать библиотеку *xgboost* вместо соответствующей реализации из *scikit-learn*, поскольку реализация *xgboost* намного более быстрая и масштабируемая.

Алгоритмы *RandomForestClassifier*, *RandomForestRegressor* и *xgboost* также позволяют выполнять отбор признаков в случае разреженных данных.

Другой популярной техникой отбора неотрицательных признаков является отбор на основе критерия хи-квадрат (chi-squared). Реализация этого метода также доступна в библиотеке *scikit-learn*.

В коде выше мы применяем класс *SelectKBest* совместно с критерием хи-квадрат (*chi2*), чтобы отобрать 20 лучших признаков. Количество отбираемых признаков, по сути, является гиперпараметром, который необходимо оптимизировать, чтобы улучшить результат модели.

Не забывайте сохранять все преобразователи, которые вы применяли к обучающему набору. Они понадобятся, чтобы оценить модели на валидационном наборе.

Следующим шагом является выбор алгоритма машинного обучения и настройка гиперпараметров.

В общем случае, выбирая алгоритм машинного обучения, необходимо рассмотреть следующие варианты:

* Классификация:
  + Случайный лес (random forest).
  + Градиентный бустинг (gradient boosting).
  + Логистическая регрессия (logistic regression).
  + Наивный Байес (naive Bayes).
  + Метод опорных векторов (support vector machine).
  + Метод k ближайших соседей (k-nearest neighbors).
* Регрессия:
  + Случайный лес (random forest).
  + Градиентный бустинг (gradient boosting).
  + Линейная регрессия (linear regression).
  + Гребневая регрессия (ridge regression).
  + Lasso-регрессия (lasso regression).
  + Метод опорных векторов (support vector machine).

Какие же параметры следует оптимизировать? Как подобрать оптимальные значения параметров? Это наиболее распространенные вопросы, возникающие на этапе настройки гиперпараметров. Ответы на эти вопросы можно получить, только приобретая опыт работы с различными алгоритмами и комбинациями параметров на различных наборах данных. Следует также отметить, что не все опытные специалисты готовы поделиться своими секретами. К счастью, я имею достаточно большой опыт и охотно покажу вам свои наработки.

В таблице ниже представлены основные гиперпараметры каждого алгоритма и диапазоны их оптимальных значений.

Метка *RS\** в таблице означает, что невозможно указать оптимальные значения и следует выполнить случайный поиск (random search).

Еще раз напомню, не забывайте сохранять все примененные преобразователи:

И не забывайте применять их к валидационному набору:

Рассмотренный нами подход и основанный на нем фреймворк продемонстрировали хорошие результаты на большинстве наборов данных, с которыми мне приходилось работать. Безусловно, при решении очень сложных задач, эта методика не всегда дает хороший результат. Ничто не совершенно, но, обучаясь, мы совершенствуемся. Точно так же, как это происходит в машинном обучении.