Model Training and Improvement

За статиите - Две доказано добри научни статии и да ги докарам във вид, който е лесен за разбиране дори от човек със слаба представа от тази материя!!!! Peer review, репродуциране на статиите. Резюме на статията (Jupyter notebook)

1. Train-test split

Train-test split (разделяне на тренировъчни и тестови данни) е основна практика в машинното обучение, която включва разделяне на наличния набор от данни на две отделни части:

1. **Тренировъчен набор (Train set):**

- Използва се за обучаване на модела.
- Моделът "учи" от тези данни, за да открие модели, зависимости и връзки между входните характеристики и целевата променлива.

2. **Тестов набор (Test set):**

- Използва се за оценка на представянето на вече обучен модел.
- Данните в този набор не се използват по време на обучението, което позволява обективна оценка на способността на модела да се справя с нови, невиждани данни.

1.1 Защо е важно да се прави train-test split:

- **Предотвратяване на Overfitting:** Ако моделът бъде обучен и тестван върху едни и същи данни, той може да научи специфичните характеристики на тези данни, вместо да обобщава общите модели. Това води до лошо представяне при работа с нови данни.
- **Обективна оценка:** Разделянето осигурява независим набор от данни за тестване, което позволява реална оценка на ефективността на модела.

1.2 train_test_split FUNCTION

```
attributes_train, attributes_test, target_train, target_test = train_test_split(attributes, target)
```

Функцията `train_test_split` се използва за разделяне на набора от данни на две части:

- **Тренировъчен набор (Training set):** Използва се за обучение на модела.
- **Тестов набор (Test set):** Използва се за оценка на представянето на модела върху невиждани данни.

1.3 Stratify в `train_test_split`

В контекста на машинното обучение и разделянето на данните на тренировъчни и тестови множества, параметърът `stratify` в функцията `train_test_split` от библиотеката **scikit-learn** играе ключова роля за осигуряване на представителност на класовете във всяко подмножество. Нека разгледаме подробно какво представлява този параметър и защо е важен.
Параметърът `stratify` се използва за осигуряване на **стратифицирано** разделяне на данните. Това означава, че разпределението на класовете в тренировъчния и тестовия набор ще бъде подобно на разпределението им в целия набор от данни.

2. pd.get_dummies()

`pandas.get_dummies()` е функция в библиотеката `pandas`, която се използва за преобразуване на категорийни променливи в числови индикаторни (dummy) променливи. Това е важна стъпка в предварителната обработка на данни, особено когато работите с машинно обучение, тъй като повечето алгоритми изискват числови входни данни.
dataset-а трябва да бъде само от числа, когато го подаваме на sklearn

2.1 Какво прави `get_dummies`:

- Преобразува категорийни (неколичествени) данни в бинарни (0 или 1) колони.
- Създава нови колони за всяка уникална категория в оригиналната колона.
- Това позволява на моделите да обработват категорийни данни като числови входове.

2.2 Пример с код:

```
import pandas as pd

# Примерни данни
data = {
    'Продукт': ['Ябълка', 'Банан', 'Ябълка', 'Киви', 'Банан'],
    'Цвят': ['Червен', 'Жълт', 'Червен', 'Зелен', 'Жълт'],
    'Количество': [10, 15, 10, 5, 15]
}

df = pd.DataFrame(data)
df
```

Оригинален DataFrame:

	Продукт	Цвят	Количество
0	Ябълка	Червен	10
1	Банан	Жълт	15
2	Ябълка	Червен	10
3	Киви	Зелен	5
4	Банан	Жълт	15

Използване на `pd.get_dummies`:
Нека преобразуваме колоната `Цвят` в dummy променливи:

```
# Преобразуване на категорийната колона 'Цвят' в dummy променливи
df_dummies = pd.get_dummies(df, columns=['Цвят'])
df_dummies
```

DataFrame след прилагане на get_dummies:

	Продукт	Количество	Цвят_Жълт	Цвят_Зелен	Цвят_Червен
0	Ябълка	10	False	False	True
1	Банан	15	True	False	False
2	Ябълка	10	False	False	True
3	Киви	5	False	True	False
4	Банан	15	True	False	False

Обяснение:

1. **Оригиналният DataFrame** съдържа категорийна колона `Цвят` с три уникални стойности: `'Червен'`, `'Жълт'` и `'Зелен'`.
2. `pd.get_dummies` преобразува тази колона в три нови бинарни колони:
   - `Цвят_Червен`: 1 ако цвятът е червен, иначе 0.
   - `Цвят_Жълт`: 1 ако цвятът е жълт, иначе 0.
   - `Цвят_Зелен`: 1 ако цвятът е зелен, иначе 0.
3. **Останалите колони** (`Продукт` и `Количество`) остават непроменени, освен ако не бъдат специфицирани за преобразуване.

2.3 Допълнителни опции:

1. **Дропване на първата категория (`drop_first=True`):** Това предотвратява проблема с мултиколинейността, като премахва една от dummy променливите. Например, ако `drop_first=True`, само две от трите категории ще бъдат представени с dummy променливи.

```
df_dummies_drop = pd.get_dummies(df, columns=['Цвят'], drop_first=True)
df_dummies_drop
```

DataFrame с drop_first=True:

	Продукт	Количество	Цвят_Зелен	Цвят_Червен
0	Ябълка	10	False	True
1	Банан	15	False	False
2	Ябълка	10	False	True
3	Киви	5	True	False
4	Банан	15	False	False

В този случай, ако и двете dummy променливи са 0, това означава, че цветът е първата категория (`'Жълт'`).

3. OneHotEncoder()

`OneHotEncoder()` е инструмент от библиотеката scikit-learn, използван за преобразуване на категориални (номинални) характеристики в числов формат. Той преобразува всяка уникална категория в отделен бинарен (0 или 1) стълб. Това е особено полезно за алгоритми за машинно обучение, които работят с числови данни и не могат директно да обработват категориални характеристики.

4. OneHotEncoder() vs get_dummies()

Основни разлики между `OneHotEncoder` и `pd.get_dummies`:

4.1 Интеграция с машинно обучение (ML) пайплайни

- **OneHotEncoder**:
  - Част от **scikit-learn**, което го прави лесно интегрируем в ML пайплайни.
  - Може да бъде използван заедно с други трансформери и модели в scikit-learn, което улеснява автоматизацията на процеса на обработка на данни и обучение на модели.
- `pd.get_dummies`:
  - Функция на **pandas**, която е по-подходяща за бързо преобразуване на данни преди анализ или визуализация.

- Не е директно интегриран в scikit-learn пайплайни, което може да изисква допълнителни стъпки за включване в ML процеси.

4.2 Обработка на неизвестни категории

- **OneHotEncoder**:
  - Поддържа параметъра `handle_unknown`, който позволява обработка на категории, които не са били срещани по време на обучението (например, игнориране или присвояване на специфична стойност).
  - Това е особено полезно при трансформиране на нови данни, съдържащи непознати категории.
- `pd.get_dummies`:
  - Не поддържа директно обработка на неизвестни категории при трансформиране на нови данни.
  - Ако новите данни съдържат категории, които не са били присъстващи в оригиналния DataFrame, те няма да бъдат обработени правилно без допълнителни стъпки.

4.3 Изходен формат

- **OneHotEncoder**:
  - Може да върне разредена матрица (sparse matrix), което е по-ефективно по отношение на паметта при големи набори от данни с много категории.
  - Подходящо за модели, които могат да работят директно с разредени матрици (например, логистична регресия, линейни модели).
- `pd.get_dummies`:
  - Винаги връща плосък (dense) DataFrame, което може да заема повече памет при големи набори от данни.
  - Подходящо за по-малки до средни набори от данни и за ситуации, когато разредените матрици не са необходими.

-"sparse matrix" е вид матрица, в която повечето елементи са нули или липсващи стойности. Това е противоположността на плътната матрица (dense matrix), където повечето елементи са различни от нула.

4.4 Производителност и мащабируемост

- **OneHotEncoder**:

    - По-подходящ за големи набори от данни, особено когато се използват разредени матрици.
    - По-ефективен при обработка на данни в рамките на ML пайплайни.
- `pd.get_dummies`:
    - Подходящ за по-малки до средни набори от данни.
    - Може да бъде по-бавен и по-малко ефективен при много големи или сложни набори от данни.

4.5. Заключение

Изборът между **OneHotEncoder** и `pd.get_dummies` зависи от конкретните нужди на вашия проект:

- **Изберете OneHotEncoder**, ако работите върху машинно обучение проекти, които изискват интеграция в scikit-learn пайплайни, обработка на неизвестни категории, или ако работите с големи и сложни набори от данни.
- **Изберете** `pd.get_dummies`, ако имате нужда от бързо и лесно преобразуване на категориални данни за анализ, визуализация или за подготовка на данни, които няма да се използват директно в scikit-learn модели.

5. Pipeline

`Data Pipeline` (поток от данни) в контекста на машинното обучение (ML) представлява последователност от стъпки или процеси, които обработват и трансформират данните от източника им до финалния модел за машинно обучение. Целта на data pipeline е да осигури ефективно, надеждно и мащабируемо обработване на данните, което да поддържа създаването и поддържането на ML модели.

5.1 Защо Data Pipeline е важен в ML?

- **Ефективност и автоматизация:** Автоматизираните потоци от данни позволяват бързо и последователно обработване на големи обеми данни.
- **Надеждност и проследяемост:** Осигурява възпроизводимост на процесите и лесно проследяване на източниците и трансформациите на данните.
- **Мащабируемост:** Позволява обработка на нарастващи обеми данни без загуба на производителност.
- **Качество на данните:** Подобрява качеството на данните, което е критично за точността и ефективността на ML моделите.

- **Бързо разгръщане на модели:** Позволява по-бързо обновяване и интеграция на нови модели в продукционната среда.

!!! Добре е в края на pipelin-а да имаме `estimator - (оценител)` !!!

5.2 `pipeline.steps`

Атрибут в `scikit-learn`, който описва стъпките в **Pipeline** обект. Всяка стъпка представлява трансформер или оценител, който изпълнява специфична функция в процеса на машинно обучение.

Структура

- **Формат:** `pipeline.steps` е списък от кортежи, всеки от които съдържа:
  - **Име на стъпката** (стринг)
  - **Трансформер или оценител** (обект, реализиращ методите `fit`, `transform` и/или `fit_predict`)

6.ColumnTransformer()

6.1 Определение

`ColumnTransformer` е мощен инструмент в библиотеката scikit-learn, който позволява прилагането на различни трансформации върху различни колони (характеристики) на вашия набор от данни. Това е особено полезно, когато работите с хетерогенни данни, съдържащи както числови, така и категориални променливи.

С `ColumnTransformer` можем да създадем предварителни обработващи стъпки (preprocessing steps) за различните типове данни в единен pipeline, което улеснява целия процес на моделиране и подобрява повторяемостта на кода.

6.2 Параметърът `remainder`

Определя как да се обработват колоните, които **не са изрично посочени** в трансформерите. Това е полезно, когато искате да прилагате трансформации само на определени колони, докато останалите да бъдат оставени непроменени или обработени по друг начин.

Възможни стойности за `remainder`:

1. `'drop'` (по подразбиране): Оставените колони се игнорират и **не** се включват в резултатния набор от данни.
2. `'passthrough'`: Оставените колони се оставят **непроменени** и се включват в резултатния набор от данни.
3. **Всякакъв друг трансформър**: Можете да зададете собствен трансформър за обработка на оставените колони.

7. FunctionTransformer()

7.1 Определение

`FunctionTransformer` е трансформатор от библиотеката **scikit-learn**, който позволява прилагането на потребителски функции към данни вътре в конвейери (pipelines) за обработка на данни. Това е особено полезно, когато е необходимо да се интегрират собствени преобразувания на данни в стандартния процес на машинно обучение.

7.2 Пример за използване на `FunctionTransformer`

Нека разгледаме пример, в който искаме да приложим логаритмично преобразуване към числови данни преди обучението на модел. Това може да бъде полезно, например, за обработка на характеристики с експоненциално разпределение.

```
def log_transform(X):
        return np.log(X + 1) # Добавяме 1, за да избегнем log(0)

log_transformer = FunctionTransformer(log_transform, validate=True)

pipeline = Pipeline([
        ('log_transform', log_transformer),
        ('linear_regression', LinearRegression())
])
```

8. Запазване на модел във файл - pickle

8.1 Определение

Модулът `pickle` в Python позволява *сериализиране* и десериализиране на обекти, което е полезно за запазване на модели или данни във файл, както и за последващото им възстановяване. Това е подходящо, например, когато искате да запазите обучен модел и след това да го заредите отново без нужда от повторно обучение.

Сериализация е процесът на преобразуване на обект (например променлива, структура от данни или модел) в формат, който може да бъде съхранен (например във файл) или предаден през мрежа, а след това възстановен обратно до оригиналната си форма.

8.2 pickle.dump()

`pickle.dump(object, file)` — сериализира обекта и го записва във файл.

```
import pickle

# Примерен обучен модел (например от sklearn)
model = ...  # Това е моят модел

# Запазване на модела във файл
with open('model.pkl', 'wb') as file:
    pickle.dump(model, file)
```

8.3 pickle.load()

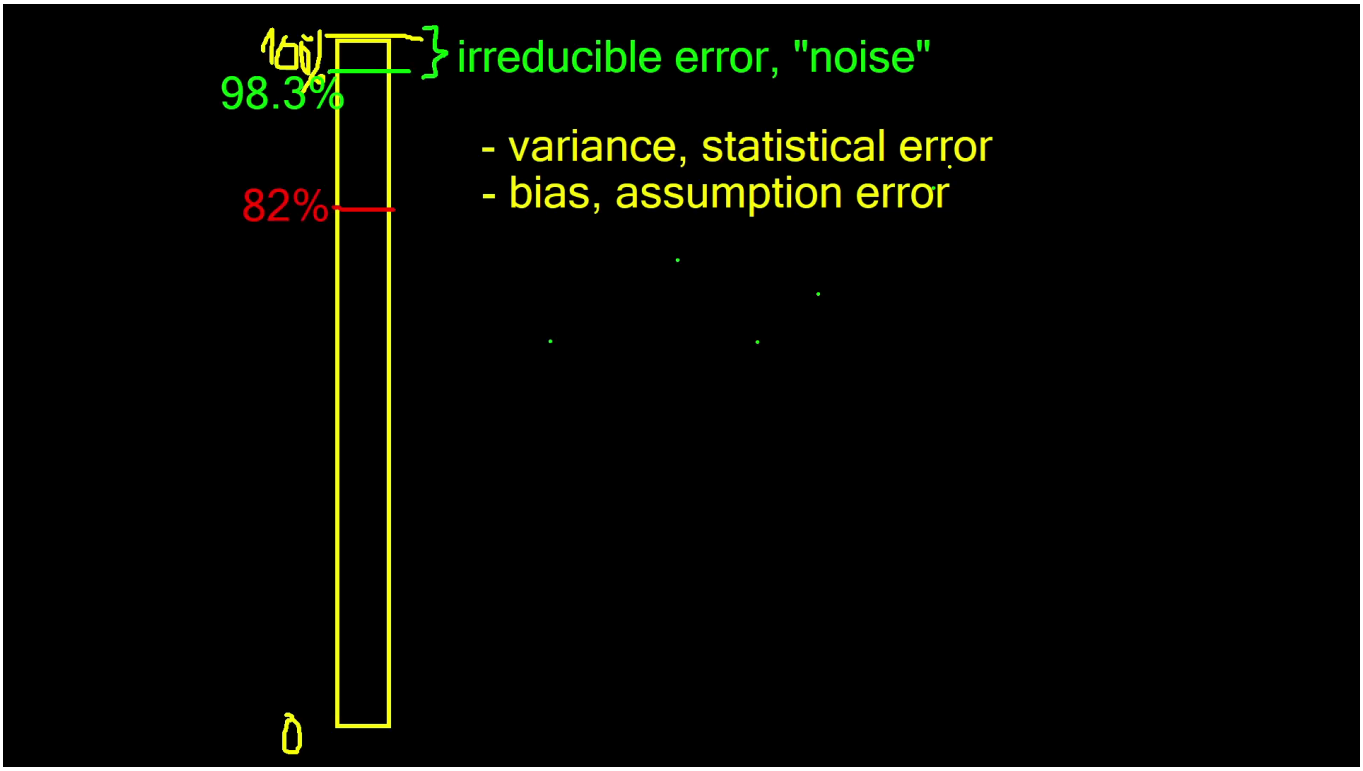
`pickle.load(file)` — десериализира обект от файл и го връща в Python.

```
import pickle

# Зареждане на модела от файл
with open('model.pkl', 'rb') as file:
    loaded_model = pickle.load(file)

# Сега мога да използвам заредения модел
```

9. Грешки в един модел

9.1 Пример

Имаме метрика на модел, който постига 82%. Дали той може да постигне 100% - не съвсем. В данните има нещо, което е непостижимо, тоест неизбежна грешка или още - шум (irreducible error, "noise"). Останалото (до 98,3%) е друг вид грешка. Случайна грешка (variance, statistical error) - това са случайности в данните, които не позволяват да ги очакваме. Например - тренирали сме с някакви данни и получаваме данни, които са малко по-различни, моделът ни не ги познава съвсем перфектно. Другият вид грешка тук се нарича bias, assumption error - грешка от предположения. Например разглеждали сме определена част от данните, в която има някакви проблеми. Това е грешка в нашия модел, за която сме виновни ние.



9.2 Bias and variance

9.2.1 Определения

- **Bias:** Измерва колко добре моделът улавя основните зависимости в данните. Висок bias означава, че моделът е твърде опростен.
  Висок bias – симптоми:
  - Моделът прави **големи грешки дори върху обучителните данни**.
  - Прекалено **опростен модел** – например, използването на линеен модел, когато данните имат нелинейни зависимости.
  - **Недостатъчно напасване (underfitting)** – моделът не улавя достатъчно добре съществуващите закономерности в данните.
    Основни причини за bias:

- Избор на твърде прост модел.
- Ограничаване на броя на характеристиките (фичърите), които се използват за обучение.
- Неправилно предположение за структурата на данните (например линейна зависимост при нелинейни данни).

Пример:

- Представете си, че тренирате модел, който предсказва цените на къщи въз основа на размера на къщата. Ако използвате твърде прост модел, като линеен регресионен модел (който предполага права линия), може да имате **високо bias**. Това ще доведе до грешка, тъй като реалната връзка между размера на къщата и цената може да бъде по-сложна (например нелинейна), а моделът ви ще прави твърде опростени прогнози.

- **Variance (Варианс):** Измерва чувствителността на модела към малки промени в данните. Високият варианс означава, че моделът прекалява с напасването на обучителните данни, но не обобщава добре върху нови данни.
  Висок variance – симптоми:
  - **Прекомерно напасване (overfitting):** Моделът е толкова сложен, че улавя и шума в обучителните данни, което води до висока вариация при прогнозите върху нови данни.
  - **Отлично представяне на обучителни данни**, но **лошо представяне на тестови данни**. Това е класически знак за висок variance.
  - Прогнозите на модела са **нестабилни**: когато обучите модела на различни части от данните или на нови данни, той прави значително различни прогнози.

Причини за висок variance:

- Използване на твърде сложен модел (например полиномиални регресии с висока степен, дълбоки decision trees или прекалено сложни невронни мрежи).
- Твърде много характеристики (features) в модела, които могат да включват шумове и незначителни зависимости.
- Недостатъчно данни за обучение – сложен модел, обучен върху малък набор от данни, може лесно да започне да учи шумовете в тях.

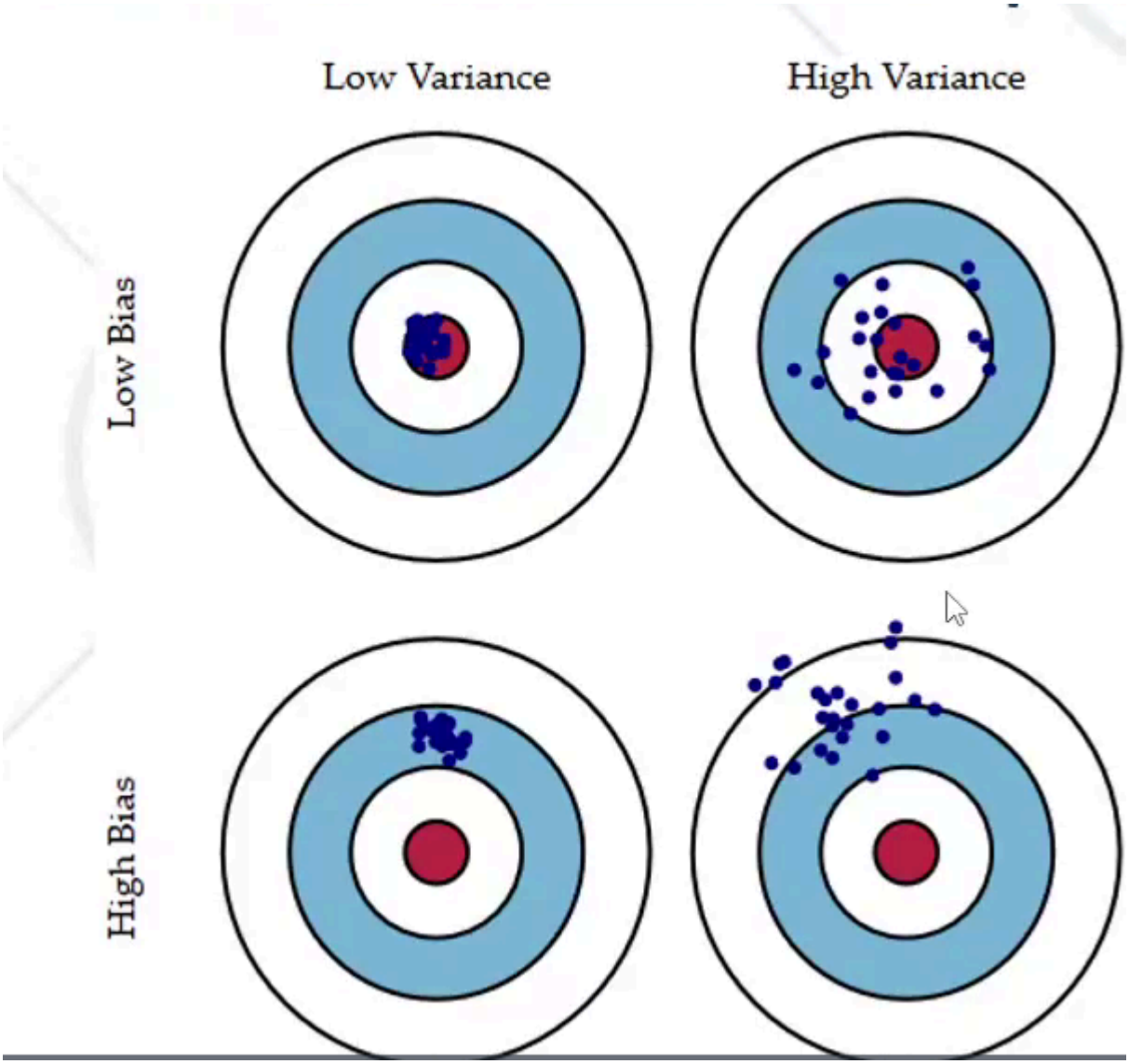
Пример:

- Представете си, че се опитвате да предскажете цените на къщите и използвате много сложен модел, например полиномиална регресия с висока степен (полином с много членове). Този модел може да се напасне много добре на обучителните данни и да постигне почти перфектни прогнози за тези данни. Но когато се приложи към нови данни (например тестови данни), прогнозите ще са

много нестабилни и с висока грешка, защото моделът е "научил" твърде много специфичности от обучителните данни, включително случайни вариации и шум.

9.2.2 Илюстрация (bias and variance)

Тази картинка илюстрира концепцията за **bias (пристрастие)** и **variance (варианс)** в контекста на машинното обучение и оценката на модели. Това са два важни аспекта, които определят колко добре моделът е способен да се учи от данните и да обобщава върху нови данни.

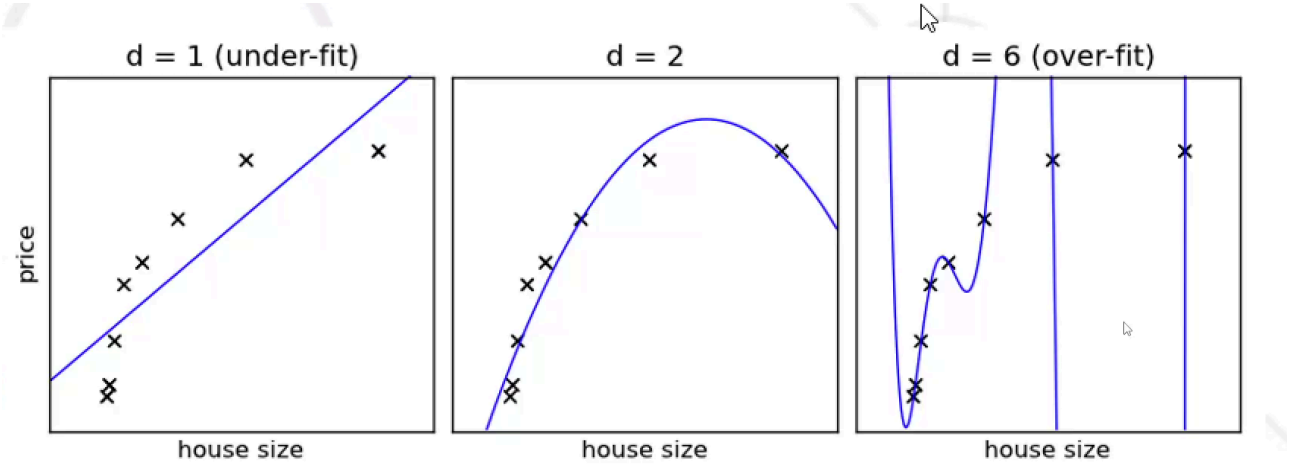


Обяснение на четирите квадранта:

1. **Горен ляв (Low Bias, Low Variance):**
   - **Нисък bias, нисък variance:** Това представлява идеалния модел. Той прави точни прогнози (центрирани върху целта) и има малка вариация между

различните прогнози (данните са много близо един до друг). Моделът е добре обучен и може да обобщава добре върху нови данни.
- Асоциация - знам къде трябва да уцеля и съм добър стрелец.

2. **Горен десен (Low Bias, High Variance):**
   - **Нисък bias, висок variance:** Моделът в този случай може да научи данните точно (има ниско пристрастие), но има висока вариация между различните прогнози. Това е знак за **overfitting (прекомерно напасване)** – моделът научава шумовете в обучителните данни, но не може да обобщава добре върху нови данни.
3. **Долен ляв (High Bias, Low Variance):**
   - **Висок bias, нисък variance:** Тук моделът е твърде опростен и има ниска вариация между различните прогнози, но те са далеч от истинската цел (висок bias). Това е пример за **underfitting (недостатъчно напасване)**, при който моделът не научава важни зависимости в данните.
4. **Долен десен (High Bias, High Variance):**
   - **Висок bias, висок variance:** Това е най-лошият сценарий. Моделът е както прекалено опростен (високо пристрастие), така и непоследователен (висок variance). Прогнозите са далеч от целта и се разпръскват значително, което показва слабо представяне и в обучителните, и в тестовите данни.
   Идеалната цел е да се намери баланс между bias и variance, така че моделът да е достатъчно сложен, за да улавя важните зависимости, но не толкова сложен, че да пренапасва обучителните данни.



10. Regularization

Регуларизацията (Regularization) е техника в машинното обучение, която се използва за предотвратяване на пренатоварване (overfitting) на моделите. Overfitting възниква, когато моделът се научи твърде добре на тренировъчните данни, включително и на шумовете

или случайните вариации, което води до лошо представяне върху нови, невиждани данни.

10.1 Защо е важна регуларизацията?

Когато създаваме модел, целта е той да обобщава добре информацията от данните, а не просто да ги запомня. Регуларизацията помага да се контролира сложността на модела, като ограничава възможностите му да се пренатоварва.

10.2 Видове регуларизация

1. **L1 Регуларизация (Lasso)**:
   - Добавя сумата на абсолютните стойности на коефициентите като наказание към функцията за грешка.
   - Това може да доведе до нулиране на някои коефициенти, което означава, че моделът може да избере само най-важните характеристики (характеристики със стойности различни от нула).
2. **L2 Регуларизация (Ridge)**:
   - Добавя сумата на квадратите на коефициентите като наказание към функцията за грешка.
   - Това води до намаляване на стойностите на коефициентите, но не ги нулира, което прави модела по-устойчив на пренатоварване.
3. **Elastic Net**:
   - Комбинира L1 и L2 регуларизацията, което позволява както намаляване, така и нулиране на коефициентите.
   - Полезен е, когато има множество свързани характеристики.

11. Logistic regression - C parameter

Параметърът **C** в **Логистичната Регресия** е важен хиперпараметър, който контролира силата на регуларизацията. Регуларизацията е техника, използвана за предотвратяване на **пренатоварване (overfitting)**, като ограничава сложността на модела. В контекста на логистичната регресия, **C** е инверсия на силата на регуляризацията, което означава, че по-малки стойности на **C** водят до по-силна регуларизация, а по-големи стойности — до по-слаба регуларизация.

11.1 Какво означава стойността на C?

- **Висока стойност на C (слаба регуларизация):**
  - Моделът се стреми да минимизира грешките на тренировъчните данни, което може да доведе до по-сложен модел.
  - По-големи стойности на коефициентите βj\beta_jβj са позволени, което може да увеличи риска от **пренатоварване**.
  - Подходящо, когато имате достатъчно данни и моделирането на по-сложни зависимости е необходимо.
- **Ниска стойност на C (силна регуларизация):**
  - Моделът позволява повече грешки на тренировъчните данни, но ограничава стойностите на коефициентите βj, правейки модела по-прост и по-генерализиращ.
  - Помага за предотвратяване на **пренатоварване**, особено когато имате малък набор от данни или много характеристики.
  - Подходящо, когато искате моделът да бъде по-устойчив на шум и вариации в данните.

11.2 Влияние на C върху модела

1. **Слаба регуларизация (високо C):**
   - Коефициентите могат да станат големи.
   - Моделът може да се адаптира прекалено добре към тренировъчните данни, включително към шумовете.
   - Риск от **пренатоварване**: Висока точност на тренировъчните данни, но ниска на тестови или нови данни.
2. **Силна регуларизация (ниско C):**
   - Коефициентите се ограничават, което води до по-прост модел.
   - Намалява риска от **пренатоварване**.
   - Може да доведе до **недотрениране (underfitting)**, ако регуларизацията е твърде силна и моделът не улови важните зависимости в данните.

11.3 Как да изберем правилната стойност на C?

Изборът на оптимална стойност за параметъра **C** е критичен за представянето на модела. Ето няколко подхода за неговото определяне:

1. **Кръстосана валидация (Cross-Validation):**
   - Разделете данните на множество подгрупи (например, 5 или 10).
   - Обучете модела на някои подгрупи и валидирайте на останалите.

- Изпробвайте различни стойности на **C** и изберете тази, която дава най-добри резултати сред всички итерации.

2. **Grid Search или Random Search:**
   - **Grid Search:** Изпитва предварително определен набор от стойности за **C** (например: 0.01, 0.1, 1, 10, 100).
   - **Random Search:** Случайно избира комбинации от стойности за тестване, което може да бъде по-ефективно при големи пространства от параметри.
   - Комбинирайте с кръстосана валидация за избор на най-добрата конфигурация.
3. **Логаритмично скалиране:**
   - Тъй като **C** може да варира в широк диапазон, често се използват логаритмични стъпки за търсене (например: 0.001, 0.01, 0.1, 1, 10, 100).
4. **Байесово оптимизиране (Bayesian Optimization):**
   - Използва вероятностни модели за избиране на следващите стойности на **C**, базирани на предишни резултати.
   - Може да бъде по-ефективно и да достигне оптималната стойност по-бързо.

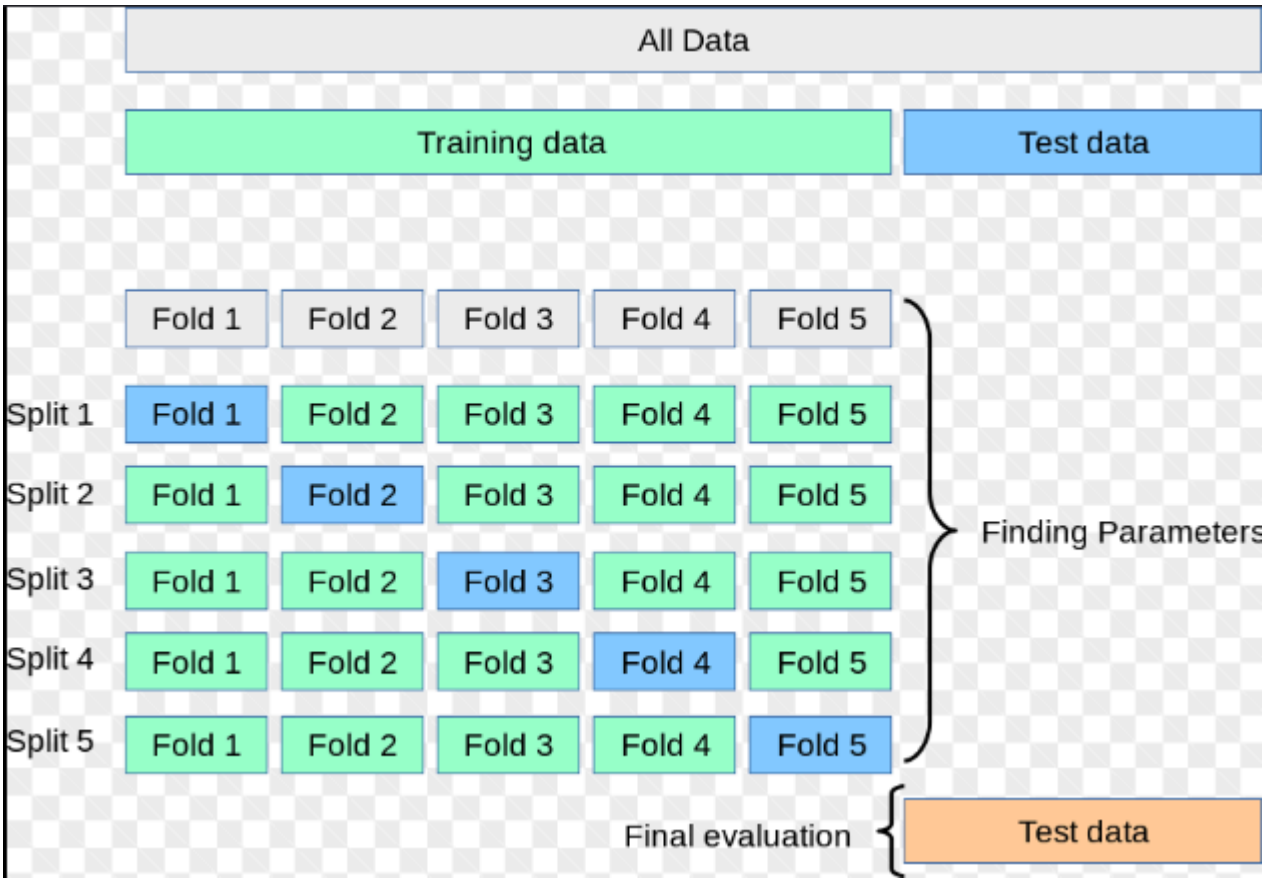
12. Метрики

В машинното обучение (ML) метриките играят ключова роля за оценка на качеството на моделите и вземане на решения относно това колко добре моделът изпълнява поставената задача. Изборът на правилните метрики зависи от типа на задачата (класификация, регресия, клъстеризация и т.н.) и специфичните изисквания на проекта.

13. ROC curve

Кривата ROC е графика, която показва съотношението между **True Positive Rate (TPR)** и **False Positive Rate (FPR)** при различни прагови стойности на класификационния модел.

14. Cross-validation



14.1 Определение

Крос валидацията (cross-validation) е техника за оценяване на качеството на моделите в машинното обучение. Тя помага да се оцени как моделът ще се представя върху нови, невиждани от него данни. Основната идея е да разделим наличните данни на няколко подмножества (наричани **folds**) и да използваме различни комбинации от тези подмножества за обучение и тестване на модела.

14.2 Основни стъпки

1. Преди да започнете с крос-валидацията, е важно да разделите вашите данни на два основни набора:
    - **Тестов Набор (Test Set):** Този набор се използва **само веднъж** за финална оценка на модела след като той е бил обучен и валидиран. Той трябва да бъде напълно отделен и невиждан по време на обучението и валидирането.
    - **Тренировъчен Набор (Training Set):** Този набор се използва за обучение и валидиране на модела. Включва всички данни, които ще бъдат използвани за настройка на хиперпараметрите и оценка на представянето чрез крос-валидация.

2. След като сте отделили тестовия набор, крос-валидацията се прилага **само върху тренировъчния набор**. Основната цел е да се оцени и оптимизира моделът, без да се докосва тестовият набор. Ето как работи процесът:

Примерен Процес с K-Fold Крос-валидация

1. **Разделяне на Тренировъчния Набор на K Фолда:**
   - Нека изберем `k=5`. Тренировъчният набор се разделя на 5 равни части (folds).
2. **Обучение и Валидиране:**
   - **Итерация 1:** Използвайте първия fold за валидиране и останалите 4 за обучение.
   - **Итерация 2:** Използвайте втория fold за валидиране и останалите 4 за обучение.
   - ...
   - **Итерация 5:** Използвайте петия fold за валидиране и останалите 4 за обучение.
3. **Събиране на Резултатите:**
   - Изчислете метриките (напр. точност, прецизност, recall) за всяка итерация.
   - Вземете средната стойност на метриките от всички итерации като окончателна оценка на модела.

14.3 Пример

При 5-кратна крос валидация (5-fold cross-validation), данните се разделят на 5 части. Моделът се обучава 5 пъти, като всеки път използва 4 части за обучение и 1 част за тестване. В края се събират резултатите от всички тестове и се изчислява средната стойност, която дава общата точност на модела.

14.4 Предимства на крос валидацията:

- **По-добра оценка на обобщаващата способност** на модела, тъй като той се тества върху различни части от данните.
- **По-малък риск от overfitting** (прекомерно нагласяване към тренировъчните данни), защото се използват различни данни за обучение и тестване.

14.5 Видове крос валидация:

1. **K-fold крос валидация:** Най-популярният метод, при който данните се разделят на k подгрупи.
2. **Leave-One-Out крос валидация (LOOCV):** Специален случай на k-fold, при който k е равно на броя на наблюденията (например, ако имаме 100 данни, правим 100 итерации с 1 наблюдение за тестване във всяка).
3. **Stratified k-fold:** Подобно на k-fold, но с допълнително условие, че при разделянето на данните се запазва съотношението на различните класове (ако става дума за класификация).

15. GridSearchCV

15.1 Определение

`GridSearchCV` (Grid Search Cross-Validation) е метод за автоматизирано търсене на най-добрите хиперпараметри за модел чрез изпробване на всички възможни комбинации от зададените стойности. Целта е да се намери комбинацията, която води до най-добро представяне на модела спрямо избраната метрика (напр. точност, F1-скор и др.)

15.2 Основни компоненти на GridSearchCV

1. **Estimator (Класификатор или Регресор):** Моделът, който искате да оптимизирате (например `RandomForestClassifier`, `SVC`, `LogisticRegression` и т.н.).
2. **Param Grid (Решетка от Хиперпараметри):** Речник, който съдържа хиперпараметрите и техните възможни стойности, които ще бъдат изпробвани.
3. **Cross-Validation (Крос-валидация):** Методът за разделяне на данните на тренировъчни и валидиращи множества по време на търсенето на хиперпараметри (напр. `KFold`, `StratifiedKFold`).
4. **Scoring (Метрика за Оценка):** Метриката, която ще се използва за оценка на представянето на модела (напр. `'accuracy'`, `'f1'`, `'roc_auc'`).