Supervised learning
[Lectures 3,4,5,6]

Data is labeled (e.g., this email is spam/this email is not)

Goals:
- *Regression*: based on past data, what sales will we have tomorrow?
- *Classification*: based on past data, will we be able to retain this client?

Techniques:
- *Regression*: linear/polynomial regression, tree-based methods
- *Classification*: logistic regression, tree-based methods

Unsupervised learning
[Lectures 8,9]

Data is unlabeled

Goals:
- *Dimension reduction*: how can I go from many features of a client to a couple of main ones that describe his/her spending?
- *Clustering*: how can I cluster clients with similar spending habits?

Techniques:
- *Dimension reduction*: PCA
- *Clustering*: k-means clustering, hierarchical clustering

Optimization

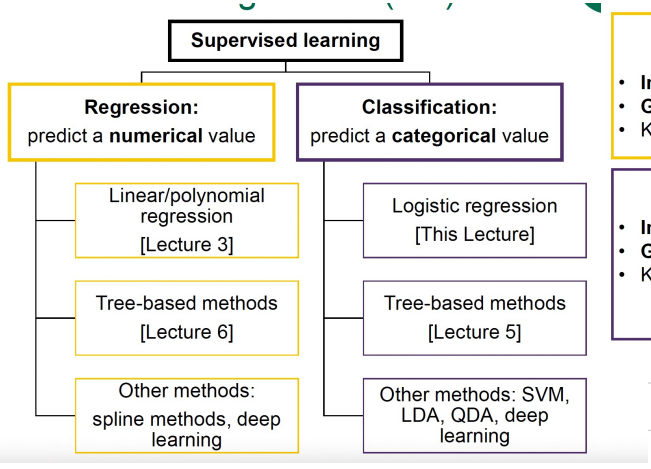
- What is an optimization problem?
- Where do they appear?

Linear optimization [Lecture 11]:

- What is linear optimization?
- How to solve a linear optimization problem in Excel?

Integer optimization [Lecture 12]:

- What is integer optimization?
- Why is it useful? Where would I use it?



Regression

- **Input :** (x_i, y_i) with x_i =features and y_i =label
- **Goal:** given x_i, predict y_i
- Key fact: y_i here is a **number**

Classification

- **Input :** (x_i, y_i) with x_i =features and y_i =label
- **Goal:** given x_i, predict y_i
- Key fact: y_i here is a **categorical variable**
  yes or no: binary classification
  meningitis or flu or covid: multinomial classification

(session 1 → 7)

Supervised Learning

- regression (value)
  - linear L.3
  - polynomial L.3
  - Tree methods L.6-7
    - version 1.0 : Regression tree
    - version 2.0 : Random forests / Boosted trees
- classification (category)
  - logistic regression
  - Tree methods
    - version 1.0 classification tree
    - version 2.0 : Random forests / Boosted trees

* **Input:** - A feature matrix X (dataframe)
→ columns = features
→ rows = observations

- Labels: y for each observation (what to predict)

Regression	Classification
= predicting a quantity	= Predicting a category

• **Regression:** Linear and Polynomial :

→ Fit a curve to your data so that it is close to your points

→ new data prediction: plug values into formula

• **Regression:** Tree methods

→ version 1.0: Regression tree : " trickle down the tree "

→ version 2.0: Random forests / Boosted trees
- better prediction
- less "jumpy" when you change the training set
- harder to interpret.

• **Classification :** Logistic regression

$$\text{Prob}_{\text{losing client}} = \frac{e^{\beta_0 + \beta_1 \cdot (\text{unhappy emails}) + \cdots}}{1 + e^{\beta_0 + \beta_1 \cdot (\text{unhappy emails}) + \cdots}}$$

→ get a probability but we want 0 or 1 answer!

→ put threshold → get category

• **Classification:** Tree methods

→ version 1.0: Classification tree : " trickle down the tree "

→ version 2.0: Random forests / Boosted trees
- better prediction
- less "jumpy" when you change the training set
- harder to interpret.

* **Data preprocessing:** → good predictions, → faster algorithms

→ need : dataframes X with numerical values and fully complete

→ so → go from categorical to numerical values
→ replace missing data

* **Training, testing, validation sets :**

→ why ? one set to train can lead to overfitting

→ Perfect model for given data but terrible for new predictions

→ **testing set:** how good the model is on new data.

How? compare predicted values with true values

regression → RMSE

classification → accuracy, confusion matrix

⚠ Testing should only be used once for your end model when you know exactly what it is

→ **validation set:** (intermediary) to pick between models

→ **training set:** create models for your data

So:
- You **train on training set** all your possible models
- You **check on validation set** which one is the best
- Once you know which one that is **you retrain on training + validation**
- Finally you **test your new model on the test set** & hand over that measure & new model!

Data pre-processing session 2

df.head(), df.info()

df.describe() → detect corrupt data / invalid values

```
Titanic[Titanic["Fare"]<0]
```
→ filter dataset

df.drop(index=number) → remove corrupt row (observations)

```
print(Titanic.shape) #gives current size of dataset
Titanic.drop_duplicates(inplace=True) # delete duplicate rows
print(Titanic.shape)
```
→ drop duplicates automatically

df.drop(columns=["column1", "column2"]) → remove features (columns)

df.isna().any() tells you which columns are empty.
df.isna().sum() tells you how many of the entries are empty.

How to detect **corrupt or invalid values** in a dataset?

For **values that are strings or objects**: use **df["column"].unique()**

Does anything seem off to you?

```
Titanic["Pclass"].unique()
array(['third', 'first', 'second'], dtype=object)

Titanic["Name"].unique().shape
(891,)

Titanic["Sex"].unique()
array(['male', 'female'], dtype=object)

Titanic["Ticket"].unique().shape
(681,)

Titanic["Cabin"].unique().shape
#some are missing: there are "nan"
(148,)

Titanic["Embarked"].unique()
array(['S', 'C', 'Q', nan], dtype=object)
```

- Two passengers share the same name
- Some passengers share ticket numbers
- There are missing values in "embarked" and "cabin"
- Valid values otherwise

How to **delete columns/features?**

- **Delete** features which are unique to each observation and bring no additional info (e.g., allocated at random)
- **Delete** features which are the same for every observation

What could we delete here?

Use df.drop(columns=["column1", "column2"])

```
Titanic=Titanic.drop(columns=["PassengerId"])
Titanic
```

	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	third	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	first	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	third	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	first	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	third	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
...										
887	first	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
888	third	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
889	first	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
890	third	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q
891	third	Johnson, Mr. William Cahoone Jr	male	19.0	0	0	LINE	0.0000	NaN	S

891 rows × 10 columns

Detect duplicates manually (see above for automated way)

```
dups=Titanic.duplicated() #checks each row of the dataset and returns TRUE or FALSE depending on whether it is a duplicate
print(dups.any()) #returns TRUE if there is any value in dups that is equal to TRUE
Titanic[dups] #returns the problematic row
```

Data imputation:

Numerical variables

Easy

Delete the row(s)/column(s) where values are missing

Replace the value with the **mean/the largest value/the smallest value**

Find the observation that is **"closest"** to it in other observations and use the value there

Find a **couple of observations** that are "close" to it and **randomly pick one of them**

Run a **regression** on rows where all the data is present and infer from it the missing values

Run a regression on rows where all the data is present and infer from it the missing values then **add noise** to the missing values

Hard

Categorical variables

Easy

Delete the row(s)/column(s) where values are missing

Create a new category: missing values

Replace the value with the **value that appears most/least**

Find the observation that is **"closest"** to it in other observations and use the value there

Find observations that are **close** to it in other observations and randomly pick one

Run a **prediction algorithm** on rows that outputs a categorical variable (Lecture 5)

Hard

Data cleansing

INSEAD

A dataset may have any of the following **issues**:

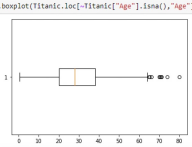
- Values that are not in the right **format** (e.g., a number that's been written as a string)
- **Invalid values** (e.g., negative values for a variable that is only positive)
- Features that bring no additional information: they contain the same **value** for all observations or different values for all observations with no useful pattern
- **Duplicate observations**
- Observations/Features that have **missing elements**

Looking for outliers:

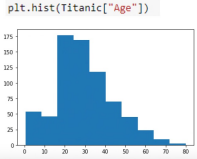
An outlier is an observation that doesn't "fit" into your dataset.

- This can be due to corrupt data, typos, or real outliers (e.g., Michael Jordan)
- Three main ways of checking for outliers: boxplots, Z-score charts (numbers above 3 or below -3), anomaly detection (unsupervised learning technique)
- Serves to flag possible outliers: area-specific knowledge to discard/keep

```
plt.boxplot(Titanic.loc[~Titanic["Age"].isna(),"Age"],vert=False)
```



```
plt.hist(Titanic["Age"])
```



Scaling and Normalizing (1/2)

INSEAD

What is **scaling/normalizing** data? Only for **numerical features.**

Makes sure that your data is on scales that are comparable.

Normalizing: operation that ensures that your data is **between 0 and 1**
Scaling: operation that ensures that the **mean** of your data is **0** and the **std dev** is **1**

Your turn!

```
from sklearn import preprocessing

X = np.array([[ 1., -1.,  2.],
...                [ 2.,  0.,  0.],
...                [ 0.,  1., -1.]])
```

```
min_max_scaler=preprocessing.MinMaxScaler()
X_minmax = min_max_scaler.fit_transform(X)
X_minmax

array([[0.5       , 0.        , 1.        ],
       [1.        , 0.5       , 0.33333333],
       [0.        , 1.        , 0.        ]])
```

Normalizing

```
X_scaled = preprocessing.scale(X)
X_scaled

array([[ 0.        , -1.22474487,  1.33630621],
       [ 1.22474487,  0.        , -0.26726124],
       [-1.22474487,  1.22474487, -1.06904497]])
```

Scaling

When should I use it?

- **Useful for certain machine learning algorithms only**.
  - Useful for regressions, PCA. Not for tree algorithms.
  - If your algorithm takes features and multiplies them by numbers etc., then chances are scaling/normalizing could improve it.
- Some use cases:
  - Some columns are **orders of magnitude different** (e.g., column A has values around 1 and column B has values around 10,000,000,000)
  - Your algorithm is returning **warning message** of the type "poor condition number"
  - The **output** you get from your algorithm is **incomprehensible** (e.g., NAs)

Data imputation (4/5)

	Sales	Size	Color
1	221	NA	Blue
2	157	Large	NA
3	NA	Medium	Red
4	50	NA	Green
5	122	Large	Red

Missing completely at random
(no pattern to the missing entries)

	Age	Mammography results
1	23	NA
2	55	Negative
3	34	Positive
4	18	NA
5	62	Positive

Missing at random
(absent entries depend on another feature)

	Weight (kgs)	Age	Diabetes
1	80	77	1
2	90	40	0
3	NA	62	1
4	50	18	0
5	NA	54	1

Missing not at random
entries are absent due to their value or a feature not accounted for)

- The methods discussed work well for **missing completely at random/missing at random**
- For **Missing not at random**: much harder, can be mitigated by adding features

Transforms & interactions

- Depending on the set-up it may be useful to **transform a feature**:
  - take powers of it
  - subtract/add a constant to it
  - divide/multiply by a constant
  - take an exponential of it or a log

Example in your homework on"Fare": converting pounds from 1912 to today's euros.

- **Feature interactions** involve adding/multiplying/dividing etc. two features together to obtain a new feature.

Example in your homework linking "ParCh" and "SibSp"

Feature engineering

Numerical ↔ Categorical (2/3)

Pclass
Second
First
Third

⇒

Pclass
2
1
3

Ordinal variables
(these entries can be ranked)

Exercise left for homework

Embarked
S
C
Q

Nominal variables
(these entries cannot be ranked)

⇒

	S	C	Q
Observation 1	1	0	0
Observation 2	0	1	0
Observation 3	0	0	1

One-Hot Encoding

⇒

	S	C
Observation 1	1	0
Observation 2	0	1
Observation 3	0	0

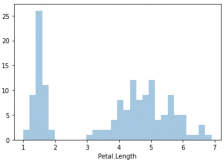
Drop redundant column

```
Titanic=pd.get_dummies(Titanic,columns=["Sex","Embarked"], drop_first=True)
```

↳ to get fully numerical dataset

Numerical ↔ Categorical (3/3)

Numerical → Categorical



Idea: use bucketization or data binning. Take average for each bucket. Number of buckets=number of categories.

Can also serve to aggregate observations.

- **Feature selection**
  Involves picking the **"right" features** for the model out of all possible features

- **Dimension reduction**
  - Involves "**merging**" features together to get as **few features** as possible to explain the variability in the data
  - Covered in unsupervised learning (Lecture 8)

Lecture 3: supervised learning: basics and regression recap

→ linear regression
→ polynomial regression

Linear regression (1/7)

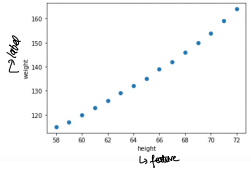
We are going to do regression on an easy dataset: **women_data**

1. Take a look at it. How many observations?
2. Plot weight as a function of height using plt.scatter.
3. Which one is the independent variable? The dependent variable?

```
women_data.head()
```

	height	weight
0	58	115
1	59	117
2	60	120
3	61	123
4	62	126
5	63	129
6	64	132
7	65	135
8	66	139
9	67	142
10	68	146
11	69	150
12	70	154
13	71	159
14	72	164

```
plt.scatter(women_data["height"],women_data["weight"])
plt.xlabel("height")
plt.ylabel("weight")
```



- Up until now, we've called **feature** any column/variable in our dataset.
- In regression, there is a special variable/feature: **the dependent variable.**
- We refer to the **dependent variable** as the **label (="special" feature).** The **independent variables** are the **features.**

Input: $(height_i, weight_i)$ pairs (15 of them)

Goal: find numbers $(a = intercept, b = slope)$ such that sum of residuals squared $(weight_1 - a - b \cdot height_1)^2 + \cdots + (weight_n - a - b \cdot height_n)^2$ is smallest possible

Output: Once we have (a, b), we can obtain the predicted values $weight_{new} = a + b \cdot height_{new}$

How to code this up in Python?

We use the **Scikit package:**

```
from sklearn.linear_model import LinearRegression
```

(1) First part: fitting the model, i.e., fitting the line:

```
X=women_data[["height"]]
Y=women_data[["weight"]]

lm = LinearRegression().fit(X, Y)
```

Specify what your x and y variables are

Model you're interested in

Fits that model to your data

(2) Second part: getting relevant parameters outputted

```
print("Intercept = ",lm.intercept_) # Print the resultant model intercept
print("Model coefficients = ", lm.coef_) # Print the resultant model coefficients (in order of variables in X)
print("R^2 =",lm.score(X,Y)) # Print the resultant model R-squared
```

See everything you can get in the documentation:
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

(3) Third part: getting the predictions

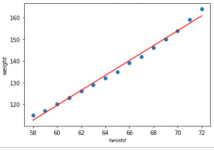
```
Y_pred=lm.predict(X)
```

Model is named "lm" and we predict from that model

(4) Fourth part (Optional): Plotting

How can we use plt.plot() and plt.scatter() to get the graph below?

print(X.loc[1])
Y_pred[1]




```
plt.scatter(X,Y)
plt.plot(X,Y_pred,c="red")
plt.xlabel("height")
plt.ylabel("weight")
```

Polynomial Regression (1/5)

We use scikit-learn again for this, but it is a bit more complicated.

Let's take a look at the code together!

(1) First part: specify the degree of interest and the features and label.

```
degree=3
X=women_data[["height"]]
Y=women_data[["weight"]]
```

(2) Second step: go from the datapoints to their "polynomial version"

For example, height datapoint 58 becomes $[1, 58, 58^2, 58^3]$

Intercept

Powers up to 3 as degree=3

```
poly = PolynomialFeatures(degree) #define the polynomial
X_poly=poly.fit_transform(X) #map all the values of X as [1,x,x^2,x^3, etc]
```

Your turn!
Double-check that this is the case by taking a look at X and X_poly.

```
[1.00000e+00, 5.80000e+01, 3.36400e+03, 1.95112e+05]
```

=1 =58 $= 58^2$ $= 58^3$

(3) Third step: fit a Linear Regression model to the polynomial datapoints.

```
polyreg = LinearRegression().fit(X_poly, Y)
```

Why linear regression?
We want to find the coefficients of the polynomial:
$c_0 + c_1 x + c_2 x^2 + c_3 x^3$

Now that we've made the datapoints polynomial, the expression above is **linear** in the coefficients ⇒ **Linear Regression**

4 Fourth step: get the coefficients (c_0, c_1, c_2, c_3) and R^2.

Your turn!

```
print(polyreg.coef_) #print these coefficients
print(polyreg.score(X_poly,Y)) #print R^2
```

```
[[ 0.00000000e+00  4.64107891e+01 -7.46184371e-01  4.25255572e-03]]
0.9997816939979363
```

5 Fifth step: Predict new points

```
y_pred=polyreg.predict(X_poly)
```

Same as before, except that we're applying it to X_poly, the transformed variables.

6 Sixth step: plot! A bit more complicated than for linear regression:

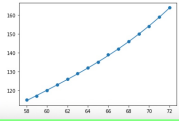
- A line is defined by two points but not a polynomial
- We need many more points to be able to generate a "pretty" curve.

```
linepoints = np.linspace(np.min(X), np.max(X), 100)
linepoints_poly=poly.fit_transform(linepoints)
linepoints_pred=polyreg.predict(linepoints_poly)

plt.plot(linepoints,linepoints_pred)
```

Here 100 points between 58 and 72!

Your turn! Try running this code!



- **R^2 increases** as the degree increases as there is more and more "freedom" in how the polynomial curve can twist.
- **The fit gets stranger and stranger** (negative values when d=12…)

Conclusions:

- This is known as **overfitting:** regression curve fits "too closely" to existing datapoints.
- Ends up **not reflecting reality** as too tailored to the dataset we have: poor prediction abilities
- Need to find **new ways of measuring** what a "good fit" is!

How good is a model at predicting?

Idea: See how good the model is at predicting sales for these new points by comparing to "real" sales.

⇒ Tells you how good the model is when faced with points it's never seen.

We predict the sales for these new time points using our **linear regression model.**

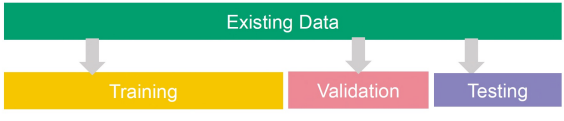
Root Mean Squared Error (RMSE) $= \sqrt{\frac{1}{n} \cdot \left((pred_1 - actual_1)^2 + \cdots + (pred_n - actual_n)^2\right)}$

If we only use the data at hand to evaluate our model, then the model can **overfit to the data**.

→ use training, testing, validation sets

Training, validation, and test sets (1/2)

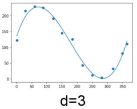
We in fact divide the existing data into **three.**



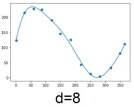
- The **training set** serves to **build your model**.
- The **validation set** serves to **select between models**.
- The **test set** is used just once to **give an indication as to how the chosen model will perform.**

Example: Polynomial regression

1. Use the **training set** to come up with polynomial regressors with different degrees.



d=3



d=8

2. Use the **validation set** to pick which degree is the best, e.g. $d = 3$.
3. Use the **testing set** to evaluate how well the model you picked would perform on new data.

These concepts in Python

We use scikit again.

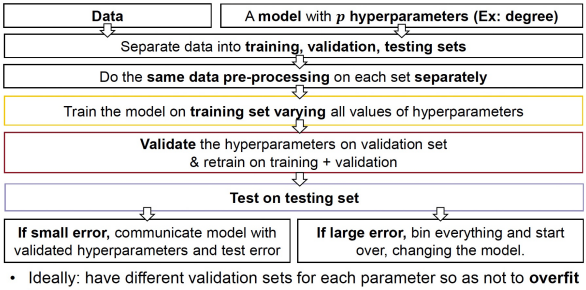
- How to **divide** a dataset into **two**:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( Xdata, ydata, test_size=0.33)
```

- How to **divide** a dataset into **three**: use the function above twice (see homework)
- How to **compute the RMSE** between real values and predicted values (example here with the linear regression model)

```
y_pred_val=lm.predict(Xval)
from sklearn.metrics import mean_squared_error
mean_squared_error(Yval,y_pred_val)**(1/2)
```

Supervised learning process



- Ideally: have different validation sets for each parameter so as not to **overfit the validation set**
- **Requires a lot of data /computation power**

Lecture 4 : supervised learning - logistic regression

→ specificity, sensitivity, accuracy
→ logistic regression
→ confusion matrix

Pitfalls in supervised ML (1/3)

Saw one already:

Overfitting

When the model is fitted too closely to the data

How do we get round it?

- This is the purpose of the validation/test sets
- If your model is performing well on the training set but not on the other sets, then you are overfitting
- If this is happening, start over

Target leakage

When the features include information that wouldn't be available at time of prediction (typically based on the actual prediction)

got_pneumonia	age	weight	male	took_antibiotic_medicine	...
False	65	100	False	False	...
False	72	130	True	False	...
True	58	100	False	True	...

Source: Kaggle

How to avoid this?

Know your data and what your features represent!

Train-test contamination

When the testing set serves to fine tune the model when it should only be used to evaluate it.

Examples of occurrence:

- If **data pre-processing** (imputation with the mean, scaling/ normalizing) is done on training and testing sets at the same time
- If **hyperparameters are tweaked** based on the testing set performance

Model contains testing set information: the performance observed on the testing set **does not reflect reality.**

From linear to logistic regression(3/3)

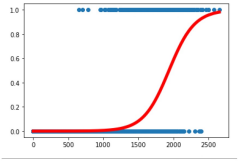
Conclusions: ↳ motivations

- We want to interpret Y_pred as the **probability** that this particular person will default.
- How to go from the **probability** to a **class**? Goal is **classification.** Let's see later!

- Probabilities are **between 0 and 1.**
- **Linear regression** does **not work** to do this: get predictions that are smaller than 0, larger than 1.
- Need to fit a function to the data that only takes on values between 0 and 1.

Logistic regression (1/4)

What happens in logistic regression? We fit a **different curve.**



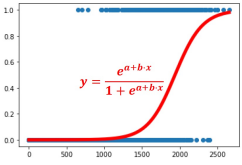
Input: datapoints (x_i, y_i)

Here x_i is the balance; $y_i = 1$ (default) or 0 (doesn't default)

We have a 10000 datapoints.

Instead of $y = a + b \cdot x$ (line), we fit a different function:

$$y = \frac{e^{a+b \cdot x}}{1 + e^{a+b \cdot x}}$$



Goal: Find numbers (a, b) such that

$\frac{e^{a+b \cdot x_i}}{1+e^{a+b \cdot x_i}}$ is as close as possible to y_i for all 10000 observations

$\hat{y}$ predicted

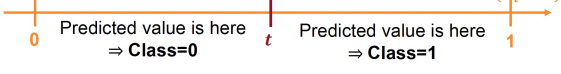
is always between 0 and 1 and can be interpreted as the **probability that observation x_i defaults (i.e., $P(y_i = 1)$).**

For each x_i,
we get a **probability (y_pred)** that the corresponding customer **will default**.

How to go from there to **classification**? i.e.,

How to **decide on labels** (defaults/doesn't default) to give to the customer?

Key idea: threshold t (by default in scikit 50%)



Logistic regression in Python

Your turn! Go through Part 3 of the notebook.

```
from sklearn.linear_model import LogisticRegression

#Logistic regression with scikit
|
X=Default[["balance"]]
Y=Default["default_Yes"]

logm = LogisticRegression().fit(X, Y) # Fit a logistic regression with vector Y as dependent and matrix X as independent

print("Intercept = ",logm.intercept_) # Print the resultant model intercept

print("Model coefficients = ", logm.coef_) # Print the resultant model coefficients (in order of variables in X)

print("R^2 =",logm.score(X,Y)) # Print the resultant model R-squared
```

```
logm.predict(X)

array([0, 0, 0, ..., 0, 0, 0], dtype=uint8)
```

Labels

```
logm.predict_proba(X)

array([[9.98694319e-01, 1.30568146e-03],
       [9.97887402e-01, 2.11259754e-03],
       [9.91405252e-01, 8.59474814e-03],
       ...,
       [9.97534484e-01, 2.46651596e-03],
       [8.83240365e-01, 1.16759635e-01],
       [9.99928552e-01, 7.14476480e-05]])
```

$P(Y = 0)$ (i.e., doesn't default)

Probabilities

$P(Y = 1)$ (i.e., defaults)

Multivariate logistic regression (1/3)

- We only used the "balance" feature here to make things visual.
- **We should be using all our features here: this is multivariate logistic regression.**
- **All the issues** that can occur with linear regression can happen here too, e.g., **multicollinearity.**
- What is the **label here**? What are the **features**?

```
Y=Default["default_Yes"] #creating the d
X=Default.drop(columns=["default_Yes"])
```

```
X.corr()
```

	balance	income	student_Yes
balance	1.000000	-0.152243	0.203578
income	-0.152243	1.000000	-0.753985
student_Yes	0.203578	-0.753985	1.000000

```
X=X.drop(columns=["income"])
X
```

We now practice the **supervised machine learning pipeline.**

- Split the data into train and test. Why is there no validation?
- **Complete questions 3 and 4 in Part 4.**

```
trainX, testX, trainY, testY = train_test_split(X, Y, test_size=0.25)
```

```
#Logistic regression with scikit

logm = LogisticRegression().fit(trainX, trainY) # Fi

print("Intercept = ",logm.intercept_) # Print the re

print("Model coefficients = ", logm.coef_) # Print t
```

```
Intercept =  [-10.7305728]
Model coefficients =  [[ 0.00569611 -0.73717829]]
```

Metrics for classification (1/4)

How should we **use our testing set?**

- For **regression**, we used the **RMSE** to compare the predicted values to the true values.
- For **classification**, our true values are "default/did not default". How do we compare these to our predicted classes?

Confusion matrix	Predicted class=0 (i.e. predicted to not default)	Predicted class=1 (i.e. predicted to default)
Actual class =0 (i.e. actual non-defaults)	True Negatives (TN)	False Positives (FP)
Actual class =1 (i.e. actual defaults)	False Negatives (FN) "bad surprises"	True Positives (TP)

Sensitivity $= \frac{TP}{TP+FN}$ (proportion of actual positives correctly identified)

Specificity $= \frac{TN}{FP+TN}$ (proportion of actual negatives correctly identified)

Accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$, **Misclassification** $= \frac{FP+FN}{TP+TN+FP+FN}$

→ better to have low FN (bad surprises) than low FP

* Confusion matrix in Python

```
Y_pred=logm.predict(testX)
from sklearn.metrics import confusion_matrix
confusion_matrix(testY,Y_pred)
```

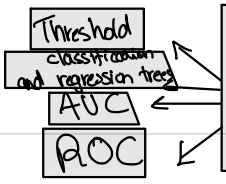
Compare the true values on the testing set to the predicted classes. **Order is very important here!!**

What do you think of the confusion matrix?

```
array([[2411,   13],
       [  49,   27]], dtype=int64)
```

We would like to have lower FN maybe, even if it possibly comes with higher FPs… How to do this?

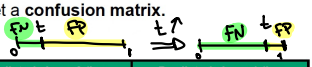
Change the threshold! (Next lecture)



• Lecture 5: supervised learning: end of logistic regression and beginning of tree methods

The impact of the threshold (1/3)

- For any **threshold**, you get a **confusion matrix.**
- **As t ↓, 1s ↑ ; as t ↑, 1s ↓ .**



	Predicted class=0 (i.e. predicted to not default)	Predicted class=1 (i.e. predicted to default)
Actual class =0 (i.e. actual no defaults)	True Negatives (TN)	False Positives (FP)
Actual class =1 (i.e. actual defaults)	False Negatives (FN)	True Positives (TP)

If t ↑, predicted 0s ↑ ⇒ FNs ↑ and FPs ↓

If t ↓, predicted 1s ↑ ⇒ FPs ↑ and FNs ↓

There is a **trade-off going on between False Negatives and False Positives** as we play around with the **threshold.**

In the last session:

```
array([[2411,   13],
       [  49,   27]], dtype=int64)
```

- we cared more about lower FNs than FPs ⇒ t ↓
- particularly important here too as a way of **correcting for imbalance of dat**

2424 labels of actual no default vs 76 labels of actual default.

The **fitting procedure** pushes towards **predicting everyone as "no default".** Why? Let's take a look at the **confusion matrix** in this case.

```
array([[2424,    0],
       [  76,    0]])
```

Accuracy=$\frac{2424}{2424+76} = 96.96\% \Rightarrow$ **Not bad!**

76 bad surprises for the bank (worst possible) & **no learning going on Blanket policy** that only works because the **data is imbalanced.**

How to set the threshold? (2/2)

- With **the model** we built using the **training set**, we can obtain the **predicted probability of default** for each observation in the **validation set.**
- We set **different thresholds**: for each threshold, we can compute a **confusion matrix** using the predicted labels on the validation set and the true labels of the validation set.
- **Complicated to make sense** of many, many **different confusion** matrices.
- How can we summarize all these confusion matrices in one place?

This is the ROC curve.

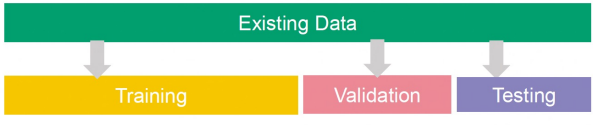
The way we set the threshold depends on which one we prefer: **do we prefer FPs or FNs?**

Activity: for each of these settings, which one of FPs or FNs would we like to lower?

- **Airport security:** "label=1" is "this person is a security threat"
- **Illness detection:** "label=1" is "this person has the illness"
- **Law:** "label=1" is "this person is guilty"
- **Detection of offensive material for removal off of social network platforms:** "label=1" is "this material is violent"

How to set the threshold? (1/2)

- As we fix **different thresholds**, we define **new models**
- **What should we do to decide which threshold to work with?**



(1) Train logistic model: obtain probabilities $P(y_i = 1)$

(2) Use validation set to pick right threshold

(4) Test the accuracy of your model on the testing set

(3) Retrain your model on training + validation. Using the threshold found, classify your observations.

How to do (2)?

The ROC curve (1/2)

On the validation set,

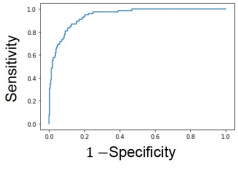
- Try **different values of the threshold between 0 and 1**
- **For each value**, write down **confusion matrix (TP,TN,FP, FN)**
- From this, for each t, can obtain

Specificity $= \frac{TN}{FP+TN}$ (proportion of true negatives correctly identified)

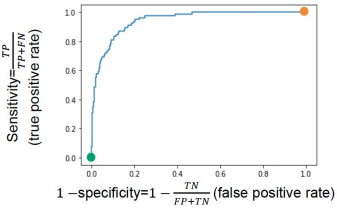
Sensitivity $= \frac{TP}{TP+FN}$ (proportion of true positives correctly identified)

- Can plot this on a curve (ROC curve):

t	1-Specificity	Sensitivity
0	0	0
0.01	0.02	0.03
…	…	…



The ROC Curve (2/2)



The points **(0,0)** and **(1,1)** always belong to the curve.

With your neighbor, think about why this happens. It might help to consider the cases $t = 0$ **and** $t = 1$.

$t = 0 \Rightarrow$ everything predicted to 1 $\Rightarrow$ FN=0, TN=0 $\Rightarrow$ Sensitivity=1, 1-Specificity=1 $\Rightarrow$ (1,1) belongs to the curve.

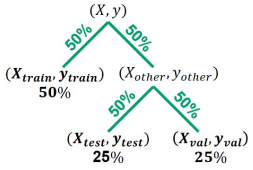
$t = 1 \Rightarrow$ everything predicted to 0 $\Rightarrow$ FP=0, TP=0 $\Rightarrow$ Sensitivity =0, 1-Specificity=0 $\Rightarrow$ (0,0) belongs to the curve.

The ROC curve in Python (1/3)

How to do this in Python?

We first split the data into test (25%), train (50%), validation (25%).

How to do this using two splits only? (also in HW)



Download MLO Lecture 5 Exercise Book & Default.csv again

Solve Part 1, Q1-5

The ROC curve in Python (2/3)

Q1.
```
Default=pd.read_csv("Default.csv")
Default=pd.get_dummies(Default,drop_first=True,columns=["default","student"])
Y=Default["default_Yes"]
X=Default.drop(columns=["default_Yes","income"])
```

Q2.
```
Xtrain, Xother, ytrain, yother= train_test_split(X,Y, test_size=0.5)

Xval, Xtest, yval, ytest=train_test_split(Xother,yother, test_size=0.5)

print(Xtrain.shape)
print(Xval.shape)
print(Xtest.shape)
```
```
(5000, 2)
(2500, 2)
(2500, 2)
```

Q3.
```
logm = LogisticRegression().fit(Xtrain,ytrain)
```

Q4.
```
y_pred = logm.predict_proba(Xval)[:,1:2]
```

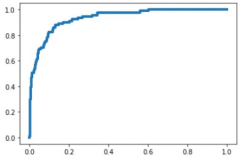
Q5.
```
roccurve=metrics.roc_curve(yval,y_pred)
roccurve
```

What does this give us?

Three arrays:
- The first one is 1-specificity
- The second one is sensitivity
- The third is the thresholds at which they've been evaluated. (Could see this via the documentation)

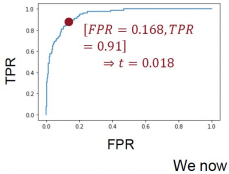
Proceed with Q6.

Q6.
```
plt.plot(roccurve[0], roccurve[1], linewidth=4)
```
```
[<matplotlib.lines.Line2D at 0x1bc7ebceb50>]
```



Using the ROC to pick a threshold (1/2)

- Trade-off between FPs and FNs: pick **best threshold for best trade-off**
- For default, we want low FN, but ok for FP to be high.
- If we want FN low, then TPR (=sensitivity= $\frac{TP}{TP+FN}$) has to be close to 1. Ok if FPR (=1-specificity=1$-\frac{TN}{FP+TN}$) not exactly 0.



Complete Part 2, Q1.
Use your roccurve to pick a threshold that works for you and your data. You may want to filter your dataset using >=, =<, &

```
((roccurve[0]<=0.3) & (roccurve[0]>=0.2)) & (roccurve[1]>=0.9)
```

We now have **our full model:**
we are going to use **logistic regression** with a **threshold of 0.018** (in my case).

Using the ROC to pick a threshold (2/2)

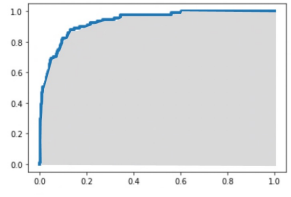
Next steps: testing our model – run at the same time as me!

1. Merge together your training and validation sets to have as much data as possible **(run Q2)**
```
Xtrain_val=pd.concat([Xtrain, Xval])
ytrain_val=pd.concat([ytrain, yval])
```
2. Re-fit your model to this new set **(run Q3)**
```
logm = LogisticRegression().fit(Xtrain_val, ytrain_val)
```
3. Go from predicted probability of defaulting to label using your threshold **(run Q**
```
Default_prob=logm.predict_proba(Xtest)[:,1] #pre
threshold=0.018
y_pred=np.where(Default_prob > threshold, 1, 0)
```
4. Deduce your confusion matrix **(run Q5)**
```
confusion_matrix(ytest,y_pred)
```
```
array([[1960,  443],
       [   5,   92]], dtype=int64)
```
```
accuracy_score(ytest,y_pred)
```
```
0.8208
```

Accuracy has gone down, but **better FPs vs FNs.**

The AUC (1/2)

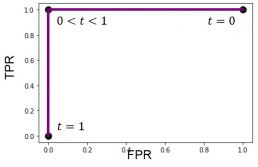
- The ROC curve can provide a **measure of how good the model is regardless of the threshold chosen, i.e.,** how good the fit of the curve is to the data.
- **Done by taking a look at the Area Under the Curve (AUC).**



In Python: try it out! (Q6)
```
roc_auc_score(ytest, Default_prob)
```

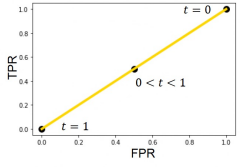
The AUC (2/2)

Best ROC curve



Why is this the best ROC curve?
See HW
AUC =1

Worst ROC curve



Why is this the worst ROC curve?
See HW
AUC=1/2

If your AUC is close to 1, great! If it's close to ½ not so great…

Wrap-up of logistic regression (1/2)

- **Logistic regression** involves **fitting a specific curve** to your training set to obtain the predicted probability of defaulting.
- To obtain labels (default/no-default), we need a **threshold.**
- This **threshold** is application-specific and can be found using **the validation set and the ROC**.
- Once you have the threshold, use the test set to obtain a **confusion matrix** and **establish how good the model is.**
- The **AUC** is another measure of goodness of fit which is **independent of the threshold.**

Wrap-up of logistic regression (2/2)

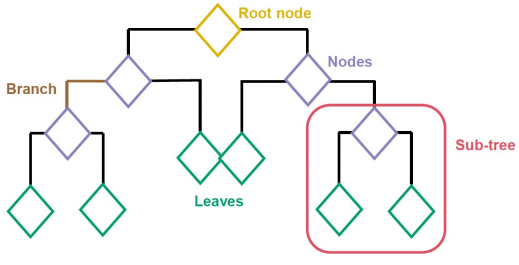
- Logistic regression as we presented it enables us to **answer yes/no questions**
- Think back to example: how to predict whether it is meningitis/flu/covid?
- This requires more than yes/no: requires us to predict **one of three categories**
- Can be done via an adaptation of logistic regression called **multinomial logistic regression**
- Not covered here as we will present a tree-based model for three outcomes or more.

Classification and Regression Trees

- **Classification and Regression Trees (CART)** takes as input **observations** for which we have **features** and **labels.**
- **Using these, CART builds a decision tree** which **branches on the features** is built.
- In other words, CART **divides the feature space** up into **boxes.**
- To pick the features to branch on, it tries to group observations that **have the same labels (or close labels)** in **the same box.**

Next time, we will see exactly **how we branch, when we stop branching,** and **how we then use the tree we have built for predictions.**

Some tree vocabulary



Branching/splitting : the act of creating new nodes
Pruning: removing leaves

Blank Page

Lecture 6: Supervised Learning: classification and regression-tree-based methods part 2

(session 8)

Unsupervised Learning

Lecture 8: introduction and PCA

Supervised learning

Supervised learning is all about **prediction.**

Predicting a **value:** **Regression**	Predicting a **category:** **Classification**
Methods: Linear/polynomial regression, CART, boosted trees, random forests	**Methods**: Logistic regression CART, boosted trees, random forests

Input to supervised learning:

(X, y)

X: feature matrix (i.e., columns are features and rows are observations)

y: label (predicted value or category) for each observation

Unsupervised learning (3/3)

"Issues" for unsupervised learning:

- No simple goal for unsupervised learning: generally used in an **exploratory data analysis**
- **No notion of "true answer"**: no immediate way of evaluating our work – often subjective
- As a consequence, no training/ testing/ validation possible

Unsupervised learning (1/3)

Unsupervised learning **cannot do** prediction. Why?

Input to unsupervised learning:

X

i.e., just the feature matrix – no labels, so no way of predicting

What does unsupervised learning do?

Tries to **draw patterns or other information** out of X

Variance and dimension reduction

- **High information in a feature = high variance of the feature.**
- **Dimension reduction:** creating a **small number** of **new features** from existing features such that **loss of information** is minimal
- **Goals:**
  - Reduce **computing time** for supervised learning routines if used down the road
  - Better **visualization**
  - New variables may **mean something:** managerial insights

How to do this dimension reduction?

A method: Principal Component Analysis

PCA (Principle component analysis)

Goal: Creating a smaller set of new features in place of the old set without losing too much information → dimension reduction

Input: n old features → PCA creates n new features → Drop some of the n new features → Output: $m < n$ new features

- ⑤ (Video to watch) Do I need to preprocess the old features before PCA?
- ① How are these features created? Why?
- ② How to do this in Python?
- ③ How do I know how many new features to drop?
- ④ How can I interpret the new features?

PCA creates **new features** by taking **linear combinations of the old features centered.**

$$z_1 = a_{01} \cdot (\text{Total Fat} - \overline{\text{Total Fat}}) + \cdots + a_{01} \cdot (\text{Vitamin C} - \overline{\text{Vitamin C}})$$

$$z_{10} = a_{91} \cdot (\text{Total Fat} - \overline{\text{Total Fat}}) + \cdots + a_{99} \cdot (\text{Vitamin C} - \overline{\text{Vitamin C}})$$

loadings

scores

	Category	Item	z1	z2	z3
0	Breakfast	Egg McMuffin	56.768221	2.172171	-39.444570
1	Breakfast	Egg White Delight	-12.352350	-10.855976	-2.597695

⇒ drop the feature with the smallest relative variance (z)

① How are these new features created?

PCA only works on numerical features.

df.head()

df.isna().any()

total variance $= \sum_{i=1}^{m}$ variance feature m

relative variance $=$ variance of feature m / total variance

	Feature 1	Feature 2
Feature 1	Variance of Feature 1	Covariance of Features 1 & 2
Feature 2	Covariance of Features 1 & 2	Variance of Feature 2

df.cov()

We now introduce 3 new features:

$$z_1 = -0.17 \cdot (\text{Vitamin C} - \overline{\text{Vitamin C}}) + \cdots + 0.81 \cdot (\text{Cholesterol} - \overline{\text{Cholesterol}})$$
$$z_2 = 0.98 \cdot (\text{Vitamin C} - \overline{\text{Vitamin C}}) + \cdots + 0.15 \cdot (\text{Cholesterol} - \overline{\text{Cholesterol}})$$
$$z_3 = 0.02 \cdot (\text{Vitamin C} - \overline{\text{Vitamin C}}) + \cdots - 0.55 \cdot (\text{Cholesterol} - \overline{\text{Cholesterol}})$$

↳ do the same as above with the z

⇒ get same total variance but different relative variance

The two datasets contain the **same amount of information but spread out in a different way among features.**

② How to do this in python?

```
from sklearn import decomposition as dcp
menu=pd.read_csv("McDonaldsMenu.csv")
menu_num=menu.drop(columns=["Item", "Category"])
```

Need **numerical** dataset.

```
pca=dcp.PCA(n_components=3)
pca.fit(menu_num)
```

Call PCA on your dataset.

Running PCA on our data to get 3 components

```
pca=dcp.PCA(n_components=3)
pca.fit(menu_num)
```

loadings

```
pca.components_
```

scores

```
data_pca = pca.fit_transform(menu_num)
data_pca
```

Variance of the new feature

```
pca.explained_variance_
```

Relative variance of the new feature

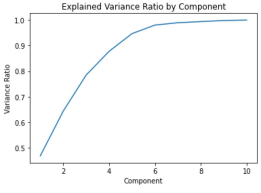
```
pca.explained_variance_ratio_
```

③ How do I know how many new features to drop?

→ With 3 features, drop the lowest relative variance.

→ For more features, see below:

To decide, we plot the **cumulative sum of relative variances** as a function of the number of components.



Pick a number of components that gives a relative variance of:

- More than 50%
- Even better: more than 80%

Here: **3-4 components would be good. Certainly not more than 5.**

④ How can I interpret the new results

- PCA can sometimes provide **additional insights** into the data.
- To do this, we need to look at the **loadings (sign and relative value).**

Let's take a look at **the loadings** for the first two new features:

	Total Fat	Saturated Fat	Cholesterol	Sodium	Carbohydrates	Dietary Fiber	Vitamin A	Calcium	Iron	Vitamin C
z1	0.414338	0.525428	0.557168	0.413492	0.094416	0.077305	0.080840	0.113909	0.154527	-0.112065
z2	0.033307	-0.105464	0.100597	0.154635	-0.024719	0.087498	0.325466	-0.168236	0.069144	0.898030

What does z_1 represent?

High fat and high salt content (=not healthy)

What does z_2 represent?

Vitamin A and Vitamin C (=healthy)

⑤ Do I need to pre-process the old features before PCA?

PCA only works on **numerical datasets** with **no missing values**

→ one hot-encoding if needed, imputation

↳ to have numerical dataset ↳ for missing values

What about **scaling/normalizing?**

Necessary when the feature units are different/incomparable.

MIM student	Height (cm)
1	178
2	163
3	181
4	175

variance = 46.687

MIM student	Height (m)
1	1.78
2	1.63
3	1.81
4	1.75

variance = 0.0046

PCA based on **variance & huge difference in variance.**
Can lead **to big differences in PCA results** if not done