Project 2 :
Titanic Dataset

M2177.005800 Basic Mathematics and Programming Practice for Machine Learning

SOHEIL HOUSNI 2024-81641
TO ALINEJAD LASHKARIANI HAMIDREZA

2024/06/11 SEOUL NATIONAL UNIVERSITY

Contents

Description of the dataset

The dataset that I used is the dataset of Titanic. The first version of the dataset comes from the Kaggle contest "Titanic: Machine Learning from Disaster" and contains the information of half of the passengers of the Titanic that sank in 1912. The datasets evolved with the work of a variety of researchers. “One of the original sources is Eaton & Haas (1994) Titanic: Triumph and Tragedy, Patrick Stephens Ltd, which includes a passenger list created by many researchers and edited by Michael A. Findlay.” Thomas Cason of Uva is one of the principal actor that updated and improved the dataset using Encyclopedia Titanica. We will use the last version of the dataset that he achieved. This version especially removed duplicate passengers, corrected errors, filled many missing ages, and added new variables.

This dataset contains the information on 1309 passengers.

There are 14 features:

- “survived”: that indicates if the passenger survived or not (1: yes, 0: no). It is the target variable.
- “pclass” :that indicates the ticket class of the passengers (1,2,3)
- “sex”
- “Age” (in years)
- “sbisp”: the number of siblings/spouses aboard
- “parch”: the number of parents/children aboard
- “ticket”: the ticket number
- “fare”: the passenger fare
- “cabin”: the cabin number
- “embarked”: the port of embarkation
- “boat”: the number of the lifeboat (only if the pasenger survived)
- “body”: the number of corps of the passenger (only if the passenger died and that his body was found out)
- “home.dest”: destination of the passenger

Importation of the dataset, pre-treatment and cleaning of data

I first imported the dataset that was an excel file. To do that, I imported the module pandas (import pandas as pd), and I stock the dataset in a variable “data”: data=pd.read_excel("titanic.xls")

To continue, before going deep into the analysis, I first did a pre-treatment and a cleaning of the data so that they can be used correctly.

I first drop all features that are less relevant for the analysis and that have less impact in the target variable results, for example the name, the number of siblings, or the cabin number:data=data.drop(["name","sibsp","parch","body","boat","home.dest","cabin", "ticket"], axis=1) (axis=1 indicates that we drop according to the columns axis: we drop the columns, so the entire features).

I then code the qualitative variables that contain strings with LabelEncoder that I first import from sklearn.preprocessing (from sklearn.preprocessing import LabelEncoder). I especially code the variable “Sex” and “Embarked”:

```
encoder=LabelEncoder()
data["sex"]=encoder.fit_transform((data["sex"]))
data["embarked"]=encoder.fit_transform((data["embarked"]))
```

I finally dropped the examples of the dataset that contain missing values:
data=data.dropna(axis=0) (axis=0 means that we drop according to the lines axis: we drop lines, so entire examples)

I finally stock the data of explenatory features and the data of the target variable into distinct data frames X and y:
X=data.drop("survived", axis=1) (means that X contains the data of our dataset except of the target variable)
y=data["survived"] (means that y is the “survived” column of the dataset and so contains the target variable)

Split of the data and analysis of the descriptive statistics

Before building our logistic regression model, we first need to split our dataset into three different datasets: the train set, the cross-validation set and the test set.

To do that, we first need to import the train_test_split method from sklearn.model_selection module (from sklearn.model_selection import train_test_split)

Then, I used np.random.seed(0) in order to control the randomity of the split to make the discussion of the results easier.

I first split in a first time the dataset into two dataset temp and test:
X_temp, X_test, y_temp, y_test =train_test_split(X,y,test_size=0.2, random_state=0)
In this way, I first obtain a fixed test set that represents 20% of total data.
Then, I did a second split by splitting the temp set in train set and cross-validation set:
X_train, X_cv, y_train, y_cv =train_test_split(X_temp,y_temp,test_size=0.25, random_state=0)
I could finally obtained the train set and the cross-validation set that represents respectively 60% and 20% of total data.

Then, I calculated the descriptive statistics (mean, standard deviation) of the entire data and of each dataset with the describe method of pandas (X.describe(), X_train.describe(), X_cv.describe(), X_test.describe()).

I also calculated the covariance matrices for each of these datasets:
np.cov(X, rowvar=False), np.cov(X_train, rowvar=False), np.cov(X_cv, rowvar=False), np.cov(X_test, rowvar=False)
I finally calculated the coefficients correlation matrices for each of these datasets:

np.corrcoef(X, rowvar=False), np.corrcoef(X_train, rowvar=False), np.corrcoef(X_cv rowvar=False), np.corrcoef(X_test, rowvar=False)

The option rowvar=False in np.cov and np.corrcoef allows to bring the precision that features in dataset are stocked in columns and not in lines: each column represents a variable and each row represents an observation.

Here are the results obtained:

Datasets	**Mean of features**				
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
X	2.206699	0.628708	29.851834	36.686080	1.548325
X_train	2.188198	0.610845	30.007842	37.243474	1.529506
X_cv	2.157895	0.665072	31.015550	37.532015	1.516746
X_test	2.311005	0.645933	28.220096	34.167963	1.636364

Datasets	**Standard deviation of features**				
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
X	0.841542	0.483382	14.389201	55.732533	0.811088
X_train	0.840083	0.487948	14.299045	55.852743	0.827062
X_cv	0.881949	0.473099	14.112568	55.594506	0.826809
X_test	0.798992	0.479378	14.852579	55.704138	0.741502

Covariance matrice of X					
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
Pclass	7.08192634e-01	5.86197730e-02	-4.97788590e+00	-2.65111893e+01	1.84446094e-01
Sex	5.86197730e-02	2.33657812e-01	4.32881470e-01	-5.06284047e+00	4.09485050e-02
Age	-4.97788590e+00	4.32881470e-01	2.07049105e+02	1.43339396e+02	-9.10218080e-01
Fare	-2.65111893e+01	-5.06284047e+00	1.43339396e+02	3.10611525e+03	-1.34564261e+01
Embarked	1.84446094e-01	4.09485050e-02	-9.10218080e-01	-1.34564261e+01	6.57863572e-01

Covariance matrix of X_train					
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
Pclass	7.05739079e-01	5.41857112e-02	-5.04660593e+00	-2.70746495e+01	1.74949427e-01

Sex	5.41857112e-02	2.38093054e-01	5.44590027e-01	-6.26348678e+00	5.14366806e-02
Age	5.04660593e+00	5.44590027e-01	2.04462695e+02	1.47951153e+02	-5.35974633e-01
Fare	-2.70746495e+01	-6.26348678e+00	1.47951153e+02	3.11952891e+03	-1.45467208e+01
Embarked	1.74949427e-01	5.14366806e-02	-5.35974633e-01	-1.45467208e+01	6.84032183e-01

Covariance of X_cv					
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
Pclass	7.77834008e-01	7.71761134e-02	-5.34381326e+00	-2.68341063e+01	2.20900810e-01
Sex	7.71761134e-02	2.23822230e-01	-2.20327804e-01	-3.42502129e+00	1.52511962e-02
Age	-5.34381326e+00	-2.20327804e-01	1.99164587e+02	1.84369385e+02	-1.26167993e+00
Fare	-2.68341063e+01	-3.42502129e+00	1.84369385e+02	3.09074911e+03	-1.29705483e+01
Embarked	2.20900810e-01	1.52511962e-02	-1.26167993e+00	-1.29705483e+01	6.83612440e-01

Covariance of X_test					
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
Pclass	6.38387928e-01	5.29536253e-02	-4.21621596e+00	-2.44109023e+01	1.66520979e-01
Sex	5.29536253e-02	2.29803092e-01	7.48174910e-01	-3.09317001e+00	3.40909091e-02
Age	-4.21621596e+00	7.48174910e-01	2.20599095e+02	8.44281047e+01	-1.50371503e+00
Fare	-2.44109023e+01	-3.09317001e+00	8.44281047e+01	3.10295094e+03	-1.05091045e+01
Embarked	1.66520979e-01	3.40909091e-02	-1.50371503e+00	-1.05091045e+01	5.49825175e-01

Coefficients correlation matrix of X					
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
Pclass	1	0.14410474	-0.41108588	-0.56525541	0.2702252
Sex	0.14410474	1	0.06223607	-0.18792965	0.10444315
Age	-0.41108588	0.06223607	1	0.17873932	-0.07799035
Fare	-0.56525541	-0.18792965	0.17873932	1	-0.29768231
Embarked	0.2702252	0.10444315	-0.07799035	-0.29768231	1

Coefficients correlation matrix of X_train					
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
Pclass	1	0.13218718	-0.42011701	-0.57702716	0.25179794
Sex	0.13218718	1	0.07805294	-0.22982554	0.1274563

Age	-0.42011701	0.07805294	1	0.18525366	-0.04532094
Fare	-0.57702716	-0.22982554	0.18525366	1	-0.31490701
Embarked	0.25179794	0.1274563	-0.04532094	-0.31490701	1

Coefficients correlation matrix of X_cv					
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
Pclass	1	0.18496426	-0.42934039	-0.54728279	0.30293459
Sex	0.18496426	1	-0.03299982	-0.13022063	0.03898947
Age	-0.42934039	-0.03299982	1	0.23499081	-0.108128
Fare	-0.54728279	-0.13022063	0.23499081	1	-0.28217692
Embarked	0.30293459	0.03898947	-0.108128	-0.28217692	1

Coefficients correlation matrix of X_test					
	Pclass	**Sex**	**Age**	**Fare**	**Embarked**
Pclass	1	0.13825328	-0.35528646	-0.54847147	0.28106988
Sex	0.13825328	1	0.10508079	-0.11583463	0.09590657
Age	-0.35528646	0.10508079	1	0.10204641	-0.13653732
Fare	-0.54847147	-0.11583463	0.10204641	1	-0.25442861
Embarked	0.28106988	0.09590657	-0.13653732	-0.25442861	1

We can observe that the desciptive statistics, whereas it is the mean or standard deviation, and the covariance matrices and coefficients correlation matrices, change really slightly from one dataset to another. This shows that the data were split correclty in a way that each dataset is representative of the full dataset. With that in consideration, we can be sure that the results of our models won't be affected by a biased distribution of data in the different train, cross-validation and test sets.

Realization of the models and analysis of the results

Basic logistic regression model

To continue, I implemented the Logistic regression model. To do that, I first imported all useful modules: from sklearn.linear_model import LogisticRegression for imported the model, from sklearn.metrics import log_loss for calculating the cost function and from sklearn.metrics import confusion_matrix, classification_report for dertemining the confusion matrix and the classification report obtained with the model.

Then, I definied the logistic regression model :model=LogisticRegression(), I fit it to my data X_temp, y_temp (as it represent 80% of data, we use X_temp as training set, and we use X_test as test set; we will use the three training, cross-validation and test set later when we will proceed with cross validation): model.fit(X_temp, y_temp), I did the prediction with the

test set (y_pred=model.predict(X_test)), and the prediction of probabilities for y=1 (y_pred_proba=model.predict_proba(X_test)[:,1]).
I could then determine the coefficients: coefficients = model.coef_ and the intercept (intercept = model.intercept_) used by the model, the cost found (log_loss_value = log_loss(y_test, y_pred_proba)); and obtain the confusion matrix (cm=confusion_matrix(y_test, y_pred)) and the classification report (classification_rep = classification_report(y_test, y_pred)) of the model.

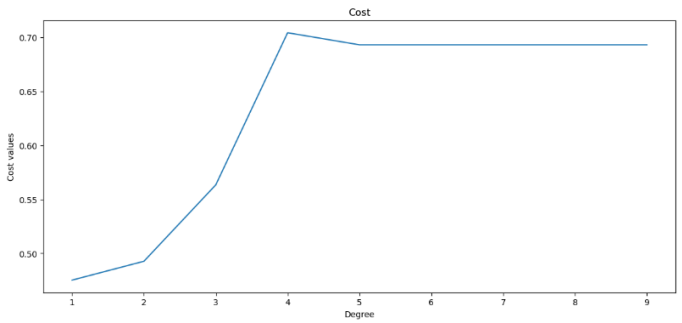
Polynomial regression model

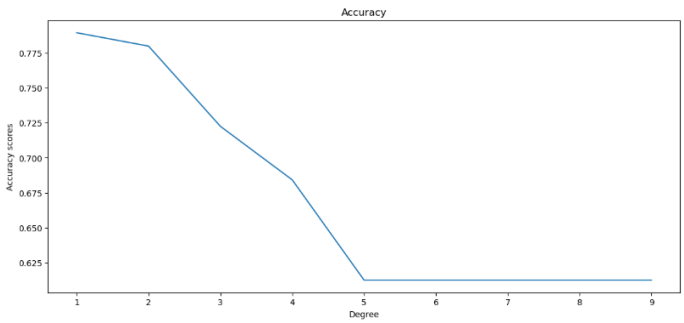
Performance of the model according to different values of degree

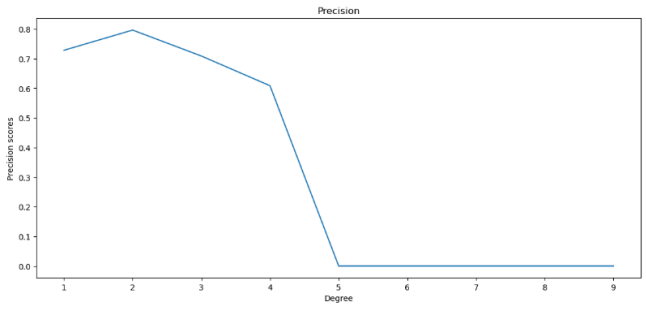
I then added to the model polynomial features. To do that, I imported the modules required (from sklearn.preprocessing import PolynomialFeatures) and the metrics for measuring performance: accuracy, f1 socre, precision and recall (from sklearn.metrics import accuracy_score,f1_score,precision_score,recall_score)

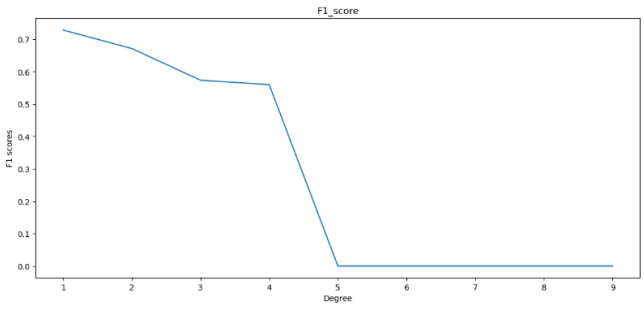
I also used np.random.seed(0) to control the randomity. The idea of the program is to evaluate the performance of the model according to different degree for polynomial features in order to find the degree with which the cost of the model is minimized.
To do that, I first created a list of values for the degree: d=[i for i in range(1,10)]. Then, I created 5 empty lists that will respectively stock the values for the cost, the accuracy, the precision, the f1 score, and the recall obtained by the model for each value of degree for polynomial features. Then, I created a loop that iterates on the values for degree (for i in d) in which we define the polynomial features (poly_features = PolynomialFeatures(degree=i)) that we then apply to X_temp and X_test (X_temp_poly=poly_features.fit_transform(X_temp) \
X_test_poly=poly_features.fit_transform(X_test)). Then, we run the model as usual.

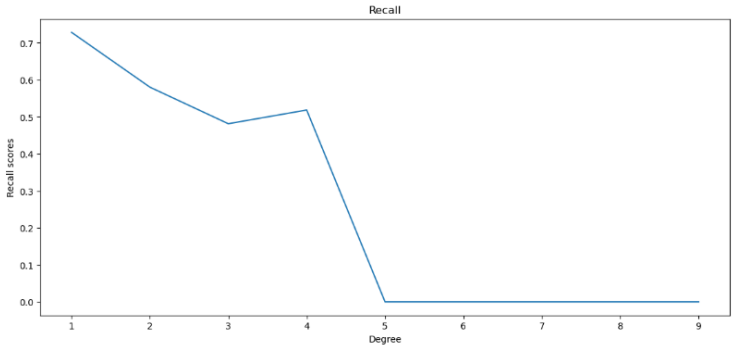
These are the graphs of the results obtained:











On the graphs, we can observe that the cost increases as the value of degree increases until 4, then it slightly decreases when the value of degree passes from 4 to 5, for finally stabilizing when the value of degree is 5.
However, what is interesting to observe on the graphs is that the precision of the model increases until the value of degree is 2 and up to 0.8. But at the same time, the cost of the model also increases.
On its side, the accuracy only decreases when the value of degree increases before stabilizing when the degree is 5.
This decrease in performance when the degree increases can be due to the fact that the model become too complex has the dataset that we use already contains 5 features (and therefore is already complex). Therefore, in the training set, the bias of the model decreases and this one is able to capture the true relationship between data in this set. However, as the model become too complex, he is also overfiting and in consequence, the variance of the model increases. That means that the model is less able to generalize and to make predictions on new data (here on the test set).

Results of the model using the best degree value

In order to find the degree with which the model has a minimum cost, I used D_cost=d[np.argmin(poly_cost_function_scores)]. In this case, it was for a degree 1.

Here are the performances of the basic logistic regression model and those of the polynomial logistic regression model with a degree 1 (for which the cost of the model is minimized).

	Cost (log-loss value)	**Accuracy**	**Precision**	**F1 score**	**Recall**
Basic logistic regression model	0.475	0.78	0.72	0.72	0.72
Polynomial logistic regression model (degree 1)	0.475	0.79	0.73	0.73	0.73

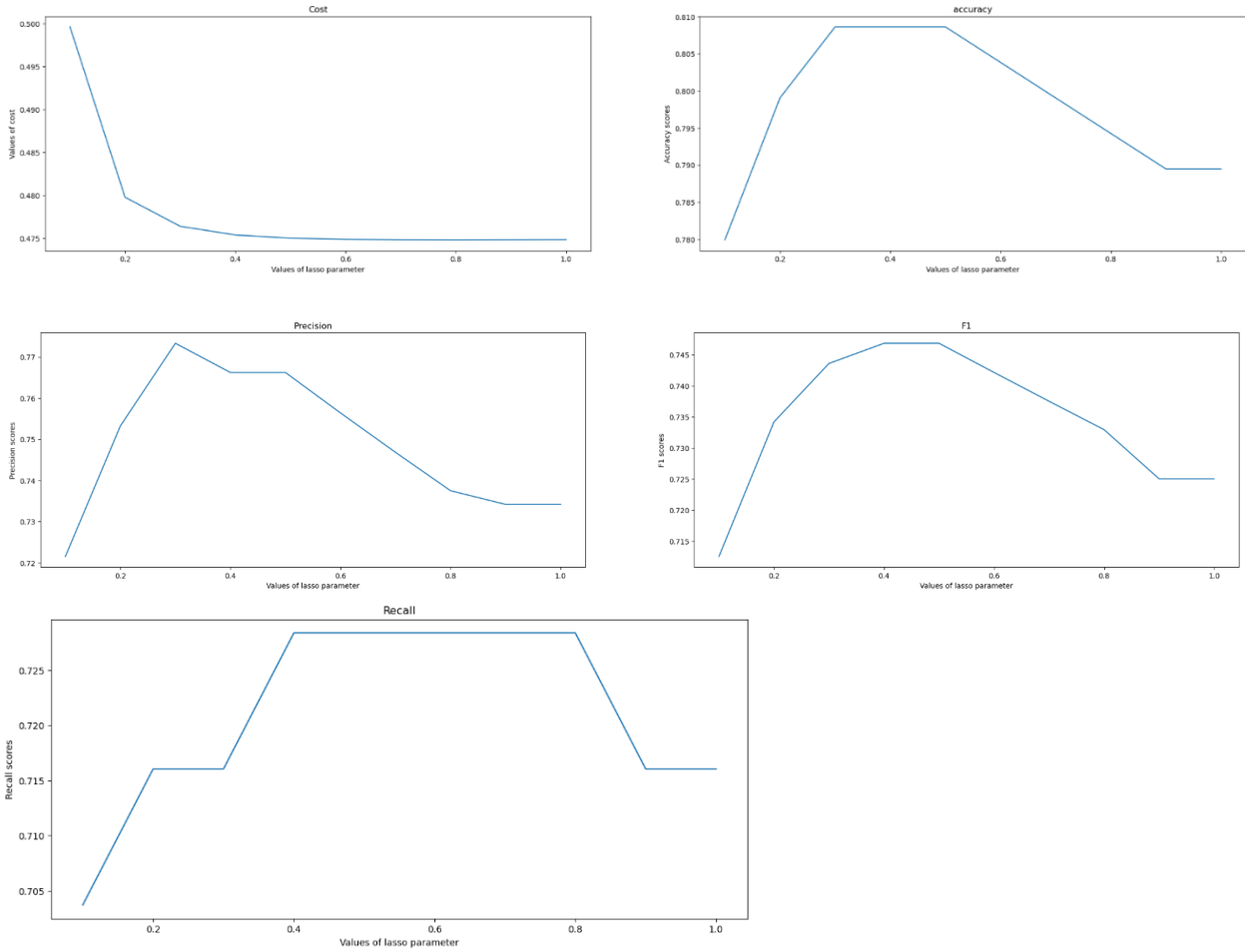
As the degree is 1, there is only a really slight improvement for the polynomial logistic regression model.

Lasso regularization

Performance of the model according to different values of lasso parameter

To continue, I apply the same reasonning for implementing the lasso regularization to our polynomial logistic regression model (that we keep with a degree of 1 as it is the degree for which the cost of the model is minimized). Therefore, like previously, I implemented the Polynomial features with a degree 1 (poly_features = PolynomialFeatures(degree=D_cost)) that I applied to X_temp and X_test (X_temp_poly=poly_features.fit_transform(X_temp) \ X_test_poly=poly_features.fit_transform(X_test)). Then, I created a list of values to try for the paramater of the lasso regularization (al=[i for i in np.linspace(0.1,1,10)]) and 5 empty lists that will respectively stock the values for the cost, the accuracy, the precision, the f1 score, and the recall obtained by the model for each value of lasso parameter. Then, I created a loop that iterates on the values for the parameter (for i in al:) in which the model is run. As we use the lasso regularization, we define the model as lasso_reg=LogisticRegression(penalty='l1', solver='liblinear', C=i) in the loop.

These are the graphs of the results obtained:



As we can see on the graphs, as the values of the lasso parameter increases, the accuracy of the model increases and stabilized before starting to decrease when the value of the parameter

is 0.5. It is the same case with the precision. This means that the bias introduced by the low values of the parameter were too strong. In consequence, when the value of the parameter increases, the bias generated by lasso regularization decreases, which allowed to reduce the cost and increase the accuracy and the precision. However, when the value of the parameter is too high, the regularization become not strong enough, which makes decrease the accuracy and the precision. Indeed, if the regularization is not strong enough, the model will overfit and the variance will increases.

Results of the model using the best lasso parameter value

In order to find the value of the lasso parameter for which the cost of the model is minimized, I used AL_cost=al[np.argmin(lasso_cost_function_scores)]. This value is 0.8. However, for this value of paramater, the accuracy is of 0.79, whereas its maximum value is arround 0.81. I compare the performance of the model using lasso regularization (that uses 0.8 as the value of the parameter) to the two previous models:

	Cost (log-loss value)	**Accuracy**	**Precision**	**F1 score**	**Recall**
Basic logistic regression model	0.475	0.78	0.72	0.72	0.72
Polynomial logistic regression model (degree 1)	0.475	0.79	0.73	0.73	0.73
Polynomial logistic regression model (degree 1) with lasso regularization (0.8)	0.4748	0.79	0.74	0.73	0.73

We can see that the model of the performance slightly increases, with a decrease in the cost and an increase in the precision. However, the value of the accuracy stayed the same, as well as the values of the F1 score and of the Recall. This only slight improvement can be explained by the fact that the Lasso regularization works better when the dataset contains a lot of useless variables. However, in this case, we already removed the useless variables.

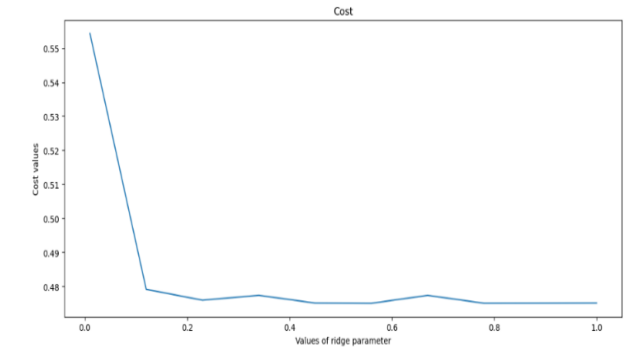
Ridge Regularization

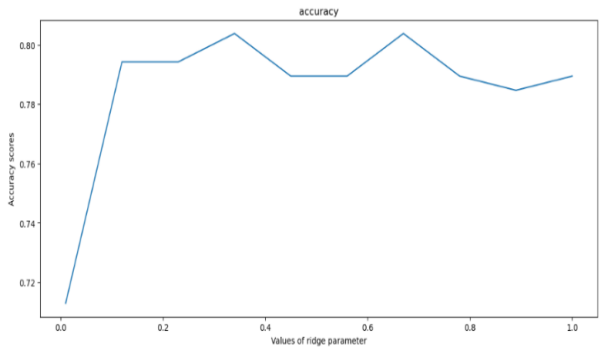
Performance of the model according to different values of ridge parameter

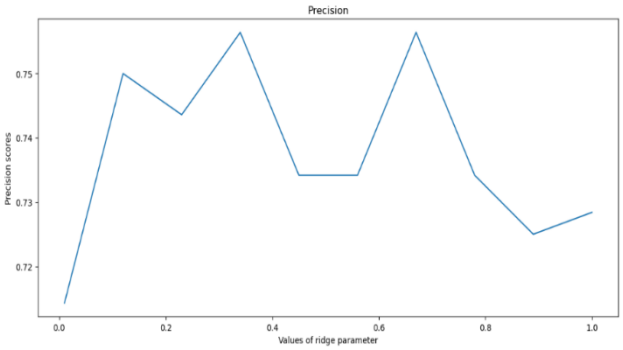
To continue, I apply the same reasonning for implementing the ridge regularization to our polynomial logistic regression model (that we keep with a degree of 1 as it is the degree for which the cost of the model in the minimum). Therefore, like previously, I implemented the Polynomial features with a degree 1 (poly_features = PolynomialFeatures(degree=D_cost)) that I apply to X_temp and X_test (X_temp_poly=poly_features.fit_transform(X_temp) \ X_test_poly=poly_features.fit_transform(X_test)). Then, I created a list of values to try for the paramater of the ridge regularization (ar=[i for i in np.linspace(0.1,1,10)]) and 5 empty lists that will respectively stock the values for the cost, the accuracy, the precision, the f1 score and the recall obtained by the model for each value of ridge parameter. Then, I created a

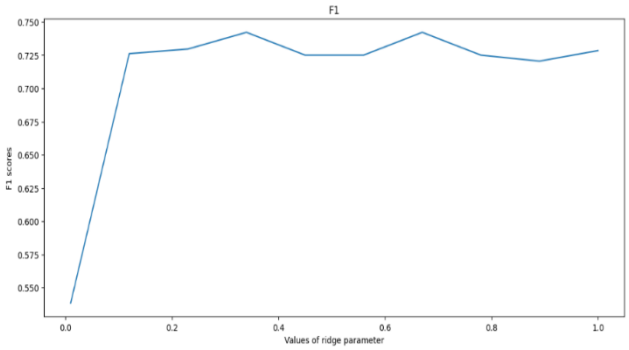
loop that iterates on the values for the parameter (for i in ar:) in which the model is run. As we use the lasso regularization, we define the model as lasso_reg=LogisticRegression(penalty='l2', C=i) in the loop.

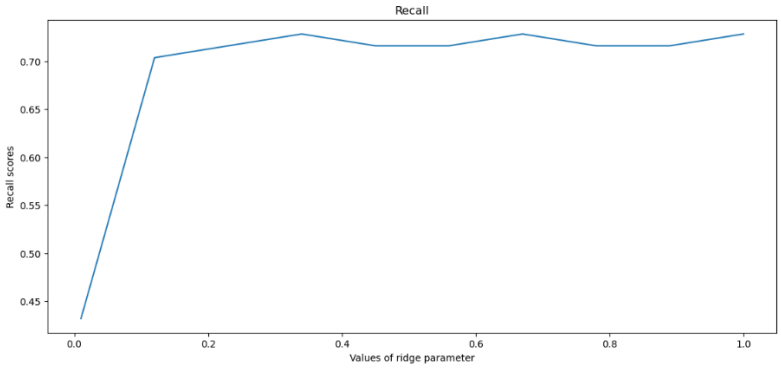
These are the graphs of the results obtained:











As we can see on the graphs, as the values of the ridge parameter increases, the cost decreases. Moreover, in contrast to the previous graph, the accuracy also increases when the value of the parameter increases, and then stabilises by slightly fluctuating. However, the precision graph fluctuates significantly. Therefore, the ridge regularization works better than the lasso regularization in this case in it can be explained by the fact that the ridge regularization works better when most of the variables are useful, which is the case here as we proceeded to feature selection.

Results of the model using the best ridge parameter value

In order to find the value of the lasso parameter for which the cost of the model is minimized, I used AR_cost=al[np.argmin(ridge_cost_function_scores)].
This value is 0.56.
I compared the performance of the model using ridge regularization (that uses 0.56 as the value of the parameter) to the three previous models:

	Cost (log-loss value)	Accuracy	Precision	F1 score	Recall
Basic logistic regression model	0.475	0.78	0.72	0.72	0.72
Polynomial logistic regression model (degree 1)	0.475	0.79	0.73	0.73	0.73
Polynomial logistic regression model (degree 1) with lasso regularization (0.8)	0.4748	0.79	0.74	0.73	0.73
Polynomial logistic regression model (degree 1) with ridge regularization (0.56)	0.476	0.8	0.76	0.74	0.72

We can see that with the ridge regularization, the cost value slightly increase. However, the accuracy, the precision and the F1 score also increases. The Recall slighlty decreased. This improvement of the performance can be due to the fact that the ridge regularization works better when most of the variables are useful.

Cross-validation

After having constructed all the different models (basic, polynomial, ridge regularization and lasso regularization), we can evaluate their generalization performance with the cross validation technique.

Basic logistic regression model

Let's start with the basic regression model.
First, I imported all the required modules: from sklearn.model_selection import KFold to use the cross validation, and
from sklearn.metrics import accuracy_score,f1_score,precision_score,recall_score to evaluate the performance of the model.

I then defined the model that I will use (model=LogisticRegression()). To continue, I created 5 empty lists that will respectively stock the scores for the cost, accuracy, precision, F1 and recall for the different iterations of the cross validation.

Then, I created a for loop: for train_index, test_index in KFold(n_splits=5, shuffle=True, random_state=0).split(X_temp,y_temp). This loop will allow to split the data set in 5 splits (which means that at each iterations, the train set will includes 4 splits and the test set will be only 1 split). .split(X_temp,y_temp) will generate the indexes for the different folds of the dataset X_temp, y_temp. Thus, train_index contains indexes of the data of X_temp that will be on the train set at each iteration, and test_index contains indexes of the data of X_temp that will be on the test set at each iteration.

Then, in the loop, we use: X_train, X_cv = X_temp.iloc[train_index], X_temp.iloc[test_index]. This lines means that, for each iterations, X_train will be the part of the dataset that contains the data that have for indexes "train_index" at this iteration, and X_cv will be the part of the dataset that contains the data that have for indexes "test_index" at this iteration. In the same way, we define y_train and y_cv with y_train, y_cv = y_temp.iloc[train_index], y_temp.iloc[test_index].

Here, the idea of the cross validation will be that, at each iterations, X_train will contain the data for k-1 folds (here it's 5-1, so 4 folds) and X_test will contain the data for 1 fold. The number of iterations is the number of pairs of train set and test set. As there are 5 splits, that means that there are 5 different pairs, so 5 iterations. At each iteration, so in the loop, we then train the model with the X_train and y_train used in the current iteration (model.fit(X_train,y_train)), we do predictions and probabilities predictions with the X_cv used at the current iteration (y_pred=model.predict(X_cv) \
y_pred_proba=model.predict_proba(X_cv)[:, 1]), we calculate the cost of the function (cost=log_loss(y_cv, y_pred_proba)) that we then add to the "cost_function_scores" list created at the beginning (cost_function_scores.append(cost)), and we can finally measure the different performances of the model , and each score will be added to its respective list. (for example with the accuracy: accuracy=accuracy_score(y_cv, y_pred) \
accuracy_scores.append(accuracy)).

When the loop is finished, we can calculate the mean of each score list to obtain the generalization performance of the model (for example with the cost: mean_cost=np.mean(cost_function_scores)).

Here are the results that we obtained:

Basic logistic regression model	**Mean cost**	**Mean accuracy**	**Mean precision**	**Mean F1 score**	**Mean Recall**
Cross-validation	0.472	0.772	0.741	0.714	0.691
Without Cross-validation	0.475	0.78	0.72	0.72	0.72

As we can observe here, the cost of the model decreased with the cross validation, passing from 0.475 to 0.472. However, the accuracy, F1 score and Recall decreased, passing respectively from 0.78 to 0.77, from 0.72 to 0.71 and from 0.72 to 0.69. Only the precision increased, passing from 0.72 to 0.74.

Polynomial logistic regression model

With the same method, I applied the cross validation to the polynomial logistic regression model. Therefore, as for the previous model, I created 5 empty lists for each of the metrics. I also use PolynomialFeatures with the degree that minimized the cost of the model (poly_features = PolynomialFeatures(degree=D_cost)) (here the degree is 1), and I tranformed X_temp with the polynomial features (X_temp_poly=poly_features.fit_transform(X_temp)). I then applied the cross validation to X_temp poly and y_temp:

```
for train_index, test_index in KFold(n_splits=5, shuffle=True,
random_state=0).split(X_temp_poly, y_temp):
    X_train_poly, X_cv_poly = X_temp_poly[train_index], X_temp_poly[test_index]
    y_train, y_cv = y_temp.iloc[train_index], y_temp.iloc[test_index]
```

Besides those changes, the method satyed the same. Here are the results that I obtained:

Polynomial logistic regression model	Mean cost	Mean accuracy	Mean precision	Mean F1 score	Mean Recall
Cross-validation	0.473	0.772	0.741	0.714	0.691
Without Cross-validation	0.475	0.79	0.73	0.73	0.73

With cross-validation	Mean cost	Mean accuracy	Mean precision	Mean F1 score	Mean Recall
Basic logistic regression model	0.472	0.772	0.741	0.714	0.691
Polynomial logistic regression model	0.473	0.772	0.741	0.714	0.691

First of all, we can observe that the changes in the results from without to with cross-validation are the same as for the previous model, which means a decrease in the cost and of the accuracy, the F1 score and the recall, but an increase in the precision. Furthermore, as the polynomial features was of degree 1, the results stay the same from basic model to polynomial model using cross-validation.

Ridge Regularization

With the same method, I applied the cross validation to the polynomial logistic regression model that uses ridge regularization. Therefore, I applied the same method as the previous model (adding also the same polynomial features part). The only change is in the loop, at the moment I defined the model: ridge_reg=LogisticRegression(penalty='l2', C=Ar_cost). This modification allows to add the ridge regularization. The values of the parameter for the ridge regularization is the one that minimizes the cost of the model (AR_cost, so 0.56).

Here are the results that I obtained:

Polynomial logistic regression model with ridge regularization	Mean cost	Mean accuracy	Mean precision	Mean F1 score	Mean Recall
Cross-validation	0.473	0.773	0.744	0.715	0.692
Without Cross-validation	0.475	0.8	0.76	0.74	0.72

With cross-validation	Mean cost	Mean accuracy	Mean precision	Mean F1 score	Mean Recall
Basic logistic regression model	0.472	0.772	0.741	0.714	0.691
Polynomial logistic regression model	0.473	0.772	0.741	0.714	0.691
Polynomial logistic regression model with ridge regularization	0.473	0.773	0.744	0.715	0.692

We can observe that from without to with cross validation, the cost decreased, but all the performance metrics also decreased.
In comparison with the 2 previous models, the cost stayed the same, but all performance metrics are better.
This shows that the ridge regularization was needed to improve the model.

Lasso Regularization

With the same method, I applied the cross validation to the polynomial logistic regression model that uses lasso regularization. Therfore, I applied the same method as the previous model (adding also the same polynomial features part). The only change is in the loop, at the moment I defined the model: lasso_reg=LogisticRegression(penalty='l1', solver='liblinear', C=AL_cost). This modification allows to add the lasso regularization. The values of the parameter for the lasso regularization is the one that minimize the cost of the model (AL_cost, so 0.8).

Here are the results that I obtained:

Polynomial logistic regression model with lasso regularization	Mean cost	Mean accuracy	Mean precision	Mean F1 score	Mean Recall
Cross-validation	0.473	0.78	0.752	0.724	0.70

Without Cross-validation	0.474	0.79	0.74	0.73	0.73

With cross-validation	**Mean cost**	**Mean accuracy**	**Mean precision**	**Mean F1 score**	**Mean Recall**
Basic logistic regression model	0.472	0.772	0.741	0.714	0.691
Polynomial logistic regression model	0.473	0.772	0.741	0.714	0.691
Polynomial logistic regression model with ridge regularization	0.473	0.773	0.744	0.715	0.692
Polynomial logistic regression model with lasso regularization	0.473	0.78	0.752	0.724	0.70

From without to with the cross validation, we can observe that the cost decreases. However, the accuracy, the f1 and the Recall also decreases. Only the precision score increases.
In comparison to the three other models, the cost stays the same. However, the accuracy, the precision, the F1 score and the Recall are better.

Therefore, we can say that the cross validation helped to reduce the cost of the model. However, it also impacted its performance, decreasing its performance metrics (except for the precision). With the cross validation, the basic logistic regression model is the model which as the minimum cost. However, the model with the lasso regularization, even if it has a slightly more important cost, performs better in all other metrics. Therefore, it is the best model using the cross-validation.

Final performance of the best model (polynomial logistic regression model using lasso regularization)

In order to evaluate the final performance of the best model, which is the one using lasso regularization, we can use the Test_set.
Here are the results obtained:

Polynomial logistic regression model with lasso regularization	**Cost**	**Accuracy**	**Precision**	**F1 score**	**Recall**
Final performances	0.477	0.78	0.752	0.724	0.70

We can see that only the cost changed, by an increase from a mean value of 0.473 to a final value of 0.477. Otherwise, the values of all other metrics stayed the same. This increase of the cost in the test set can be explained by the fact that here, the model has been trained and evaluated on less data witht the cross validation (using for both 80% of total data) than if data were splited in only two sets (of 80% and 20%). Therefore, when using the model on the test set (that represents 20% of total data), the cost increased.

Feature scaling

Results of the different models

Finally, I applied the feature scaling with the Min-max nsormalization technique. To do so, I first imported the required module (from sklearn import preprocessing).

Then, for each of the previous models, I added in the cross validation loop the scaling of the features: I first defined the scaler that I used, here the Min-max normalization (scaler = preprocessing.MinMaxScaler()), then I fit and transform my training data with it (X_train_norm=scaler.fit_transform(X_train)), and I finally transformed my validation data by conservating the parameters of the scaling resulting of the fiting with the training data (so I just transformed my validation data) (X_cv_norm=scaler.transform(X_cv)).
For the models that include polynomial features, still in the loop, I definied the polynomial features components (poly_features = PolynomialFeatures(degree=D_cost)). I then fited and transformed the scaled trained data with polynomial features (X_train_poly_norm=poly_features.fit_transform(X_train_norm)) and transformed the scaled validation data with the same parameters of polynomial features obtained with the training data (X_cv_poly_norm=poly_features.transform(X_cv_norm)). I finally used those final training and validation data to run the model. As previously, for each hyperparameter used (degree, ridge parameter and lasso parameter), I kept the value originally obtained that minimizes the cost of the function.

Here are the results that I obtained:

Basic logistic regression model	**Cost**	**Accuracy**	**Precision**	**F1 score**	**Recall**
With features scaling and cross validation	0.472	0.78	0.75	0.72	0.70
Without features scaling, but with cross validation	0.472	0.772	0.741	0.714	0.691

As for the basic logistic regression model, the cost stayed the same, but we can see that the score for each performance metric increased.

Polynomial logistic regression model	**Cost**	**Accuracy**	**Precision**	**F1 score**	**Recall**
With features	0.472	0.78	0.75	0.72	0.70

scaling and cross validation					
Without features scaling, but with cross validation	0.473	0.772	0.741	0.714	0.691

For the polynomial logistic regression model, the cost slightly decreased, and the score for each performance metric increased.

Polynomial logistic regression model with ridge regularization	**Cost**	**Accuracy**	**Precision**	**F1 score**	**Recall**
With features scaling and cross validation	0.473	0.78	0.75	0.72	0.70
Without features scaling, but with cross validation	0.473	0.773	0.744	0.715	0.692

Here again, for the model with the ridge regularization, the cost stayed the same but the performance metrics improved.

Polynomial logistic regression model with lasso regularization	**Cost**	**Accuracy**	**Precision**	**F1 score**	**Recall**
With features scaling and cross validation	0.472	0.78	0.76	0.73	0.70
Without features scaling, but with cross validation	0.473	0.78	0.752	0.724	0.70

For the lasso regularization, the cost decreased with the feature scaling, and the score for each performance metrics increased.

Final performance of the best model (polynomial logistic regression model using lasso regularization)

With feature scaling and cross-validation	Mean cost	Mean accuracy	Mean precision	Mean F1 score	Mean Recall
Basic logistic regression model	0.472	0.78	0.75	0.72	0.70
Polynomial logistic regression model	0.472	0.78	0.75	0.72	0.70
Polynomial logistic regression model with ridge regularization	0.473	0.78	0.75	0.72	0.70
Polynomial logistic regression model with lasso regularization	0.472	0.78	0.76	0.73	0.70

From this tab, we can see that the model which has the best performance is the polynomial logistic regression model with the lasso regularization.
In order to determine the final performance of this model, we need to measure the performance on the test set.

To do that, I apply the scaler and the polynomial features on the test set (keeping the parameters obtained with the fiting on train data):

```
X_test_norm=scaler.transform(X_test)
X_test_poly_norm=poly_features.transform(X_test_norm)
```

Here are the results that I obtained:

Polynomial logistic regression model with lasso regularization	Cost	Accuracy	Precision	F1 score	Recall
Final performances	0.477	0.782	0.756	0.728	0.70

Here are the results without the feature scaling.

Polynomial logistic regression model with lasso regularization	Cost	Accuracy	Precision	F1 score	Recall
Final performances	0.477	0.78	0.752	0.724	0.70

We can see that the performance of the model slightly increased with feature scaling.

Conclusion

Here is the first table of performance of the models without cross validation and feature scaling:

	Cost (log-loss value)	Accuracy	Precision	F1 score	Recall
Basic logistic regression model	0.475	0.78	0.72	0.72	0.72
Polynomial logistic regression model (degree 1)	0.475	0.79	0.73	0.73	0.73
Polynomial logistic regression model (degree 1) with lasso regularization (0.8)	0.4748	0.79	0.74	0.73	0.73
Polynomial logistic regression model (degree 1) with ridge regularization (0.56)	0.476	0.8	0.76	0.74	0.72

Here is the table of performance of the model with cross validation and feature scaling:

With feature scaling and cross-validation	Mean cost	Mean accuracy	Mean precision	Mean F1 score	Mean Recall
Basic logistic regression model	0.472	0.78	0.75	0.72	0.70
Polynomial logistic regression model	0.472	0.78	0.75	0.72	0.70
Polynomial logistic regression model with ridge regularization	0.473	0.78	0.75	0.72	0.70
Polynomial logistic regression model with lasso regularization	0.472	0.78	0.76	0.73	0.70

Therefore, we can see that the cross validation and the feature scaling allowed to improve the model, by reducing its cost. However, we can see small decreases in the performance metrics. Moreover, without the cross validation and the feature scaling, we determined that the best model is the one that employe ridge regluarization, because even if it has a slightly higher cost than the one with the lasso regularization, it outperforms all other model on performance metrics.
However, when it comes with the cross validation and feature scaling, the best model is the one which uses lasso regularization, as it has a lower cost compared to the ridge regularization, and it performs better on other performance metrics.