Lab3_Solution

September 27, 2020

0.1 Problem 2, Scraping, Entropy and ICML papers.

[ ]:
```
!pip install pdfminer.six

import os
import requests
from urllib.parse import urljoin
from bs4 import BeautifulSoup
import urllib
import io
import string
import csv
import joblib

from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from pdfminer.pdfdocument import PDFDocument
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.pdfpage import PDFPage
from pdfminer.pdfparser import PDFParser
from pdfminer.high_level import extract_text

from nltk import word_tokenize
from nltk.corpus import stopwords

import pandas as pd
import numpy as np

# Suppress warnings for readability
import warnings
warnings.filterwarnings("ignore")


url = "http://proceedings.mlr.press/v70/"

response = requests.get(url)
soup= BeautifulSoup(response.text, "html.parser")
```

```python
all_pdf_urls = [urljoin(url,link['href']) for link in soup.select("a[href$='.
 ↪pdf']")]

def convert_online_pdf_to_string(pdfurl):
    file = io.BytesIO(urllib.request.urlopen(pdfurl).read())
    output_string = io.StringIO()
    parser = PDFParser(file)
    doc = PDFDocument(parser)
    rsrcmgr = PDFResourceManager()
    device = TextConverter(rsrcmgr, output_string, laparams=LAParams())
    interpreter = PDFPageInterpreter(rsrcmgr, device)
    for page in PDFPage.create_pages(doc):
        interpreter.process_page(page)
    return(output_string.getvalue())

def convert_link_pdf_to_string(link):
    filename = link['href'].split('/')[-1]
    pdfstring = ''
    if 'sup' in filename:
        return pdfstring
    pdfurl = urljoin(url,link['href'])
    try:
        pdfstring = convert_online_pdf_to_string(pdfurl)
    except:
        print('Cannot parse {}'.format(filename))
    return pdfstring
```

```
Requirement already satisfied: pdfminer.six in /usr/local/lib/python3.6/dist-
packages (20200726)
Requirement already satisfied: sortedcontainers in
/usr/local/lib/python3.6/dist-packages (from pdfminer.six) (2.2.2)
Requirement already satisfied: chardet; python_version > "3.0" in
/usr/local/lib/python3.6/dist-packages (from pdfminer.six) (3.0.4)
Requirement already satisfied: cryptography in /usr/local/lib/python3.6/dist-
packages (from pdfminer.six) (3.1.1)
Requirement already satisfied: cffi!=1.11.3,>=1.8 in
/usr/local/lib/python3.6/dist-packages (from cryptography->pdfminer.six)
(1.14.2)
Requirement already satisfied: six>=1.4.1 in /usr/local/lib/python3.6/dist-
packages (from cryptography->pdfminer.six) (1.15.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.6/dist-
packages (from cffi!=1.11.3,>=1.8->cryptography->pdfminer.six) (2.20)
```

0.1.1 Scrape all the pdfs of all ICML 2017 papers from http://proceedings.mlr.press/v70/.

```
[ ]: #parse all pdfs to string
all_pdf_strings = []
index = 0
for link in soup.select("a[href$='.pdf']"):
    filename = link['href'].split('/')[-1]
    if 'supp' in filename:
        continue
    pdfurl = urljoin(url,link['href'])
    try:
        pdfstring = convert_online_pdf_to_string(pdfurl)
        all_pdf_strings.append(pdfstring)
#         print(str(index)+' : '+ filename)
        index += 1
    except:
        print('Cannot parse {}'.format(filename))
```

Preprocess text

```
[ ]: # Preproessing Text:
# There are several ways to preprocess text.
# Many Python libraries (nltk, scikit learn, gensim,...)
# alredy have built-in functions to do it
import string
import re
def preprocess(docs):

    #remove breaklines, convert to lowercase
    docsProc = [str.replace(docs[i], '\n', ' ') for i in range(len(docs))]
    docsProc = [u.lower() for u in docsProc]

    #remove punctuation
    docsProc = [''.join(c for c in doc if c not in string.punctuation) for doc␣
↪in docsProc]

    #remove numbers
    docsProc = [re.sub("\d+", " ", doc) for doc in docsProc]

    #trim whitespace
    docsProc = [re.sub( '\s+', ' ', doc ).strip() for doc in docsProc]

    # Vectorize text by using bag of words.
    # Notice that this function has parameters to do some of the preprocessing␣
↪above...
    from sklearn.feature_extraction.text import CountVectorizer
```

```
    #read parameters of this function for text preprocessing... stopwords,␣
↪lowercase, etc
    vectorizer = CountVectorizer(stop_words='english', lowercase =True)
    X = vectorizer.fit_transform(docsProc)

    # One way to easily explore frequency terms is converting X to a DAtaframe
    return pd.DataFrame(X.toarray(), columns = vectorizer.get_feature_names())

# Convert text to Bag of Words data frame
dtm = preprocess(all_pdf_strings)
```

1. What are the top 10 common words in the ICML papers?

[ ]:
```
def top_k_words(df, k):

    # Sum the frequency of each word over all documents
    highest_freq = df.sum(axis=0)
    highest_freq = highest_freq.sort_values(ascending=False)

    return highest_freq[0:k]

print("Top 10 words:")
top_k_words(dtm, 10)
```

```
Top 10 words:
```

[ ]:
```
cid          33434
al           14891
et           14344
learning     10952
model         7851
data          7388
algorithm     7110
set           5942
function      5471
using         5301
dtype: int64
```

0.1.2 2. Entropy

Let Z be a randomly selected word in a randomly selected ICML paper. Estimate the entropy of Z.

To estimate the entropy of Z we first need to estimate the distribution of Z.

- Each PDF is randomly selected with probability $P(\text{paper} = i) = \frac{1}{\#\text{papers}}$.
- Let N_i be the number of different words in the paper i. Then P(Z=z | paper = i) can be

estimated by the relative frequency of the word z in paper i.

$$P(Z = z|\text{paper} = i) = \frac{\text{absolute frequeny of } z \text{ in paper } i}{N_i}$$

Using the law of total probability, the marginal distribution of Z is

$$P(Z = z) = \sum_{i \in [\#\text{papers}]} P(Z = z|\text{paper} = i) P(\text{paper} = i) = \frac{1}{\#\text{papers}} \sum_{i \in [\#\text{papers}]} P(Z = z|\text{paper} = i)$$

[ ]:
```
# Probability of each paper i, P(paper=i)
prob_of_paper = 1.0/dtm.shape[0]
# Total number of words in paper i, N_i
total_words_per_paper = dtm.sum(axis=1)
# Relative Frequency of each word in each paper, P(Z=z|paper=i)
relative_freqs_per_paper = dtm.divide(total_words_per_paper, axis=0)

# Marginal Probability for each word P(Z=z)
P_z = prob_of_paper*relative_freqs_per_paper.sum(axis=0)

# Entropy
entropy = -P_z.multiply(np.log(P_z)).sum()
print(entropy)
```

```
8.606904135491124
```

0.1.3 3. Synthesize a random paragraph

[ ]:
```
# Produce a paragraph of length l
# param l: lenghth of the paragraph.
# param p: words distribution
def produce_paragraph(l, p):
    wordsIndex = [np.random.multinomial(1,p,1).argmax() for i in range(l)]
    return " ".join(p.index.values[wordsIndex])
produce_paragraph(100, P_z)
```

[ ]:
```
'inner bengio cid allow com language re ect term hys product structure
proceedings sci general ratio cid et huang longer lationship rn  rst razen time
mode machine problem update corre phaselift let models extensions recent global
language mod matrix usercf data kruskal domains action maximum tells unbounded
introduction transactions local sentation bit microsoft ssc solution ensuring
 nal combinations matrix input difference data bt integral allows visual
conditions nal essentially ple dbscan karpinski details thaler sutton lower
structure classical sequence shift production inference enjoyed reconstruction
simply  x set maha number shown recommended xcid gibbs knearest node
generalization por function jagadeesh parti trained'
```

0.1.4 Problem 2: Starting in Kaggle.

Soon, we are opening a Kaggle competition made for this class. In that one, you will be participating on your own. This is an intro to get us started, and also an excuse to work with regularization and regression which we have been discussing. 1. Let's start with our first Kaggle submission in a playground regression competition. Make an account to Kaggle and find https://www.kaggle.com/c/house-prices-advanced-regression-techniques/ 2. Follow the data preprocessing steps from https://www.kaggle.com/apapiu/house-prices-advancedregression-techniques/regularized-linear-models. Then run a ridge regression using = 0.1. Make a submission of this prediction, what is the RMSE you get? (Hint: remember to exponentiate np.expm1(ypred) your predictions).

[ ]:
```
from sklearn.linear_model import Ridge, Lasso, RidgeCV, LassoCV
from sklearn.model_selection import cross_val_score
from scipy.stats import skew
import matplotlib.pyplot as plt

if not os.path.exists('./predictions'):
    os.mkdir('./predictions')

train = pd.read_csv("train.csv")
test = pd.read_csv("test.csv")
all_data = pd.concat((train.loc[:,'MSSubClass':'SaleCondition'],
                      test.loc[:,'MSSubClass':'SaleCondition']))
```

[ ]:
```
#log transform the target:
train["SalePrice"] = np.log1p(train["SalePrice"])

#log transform skewed numeric features:
numeric_feats = all_data.dtypes[all_data.dtypes != "object"].index

skewed_feats = train[numeric_feats].apply(lambda x: skew(x.dropna())) #compute␣
 ↪skewness
skewed_feats = skewed_feats[skewed_feats > 0.75]
skewed_feats = skewed_feats.index

all_data[skewed_feats] = np.log1p(all_data[skewed_feats])

all_data = pd.get_dummies(all_data)
all_data = all_data.fillna(all_data.mean())
X_train = all_data[:train.shape[0]]
X_test = all_data[train.shape[0]:]
y = train.SalePrice
```

[ ]:
```
def rmse_cv(model):
    rmse= np.sqrt(-cross_val_score(model, X_train, y,␣
 ↪scoring="neg_mean_squared_error", cv = 5))
    return rmse
```

```
cv_score = rmse_cv(Ridge(alpha=0.1)).mean()
print('CV score for a = 0.1: {0:.5f}'.format(cv_score))
```

```
CV score for a = 0.1: 0.13778
```

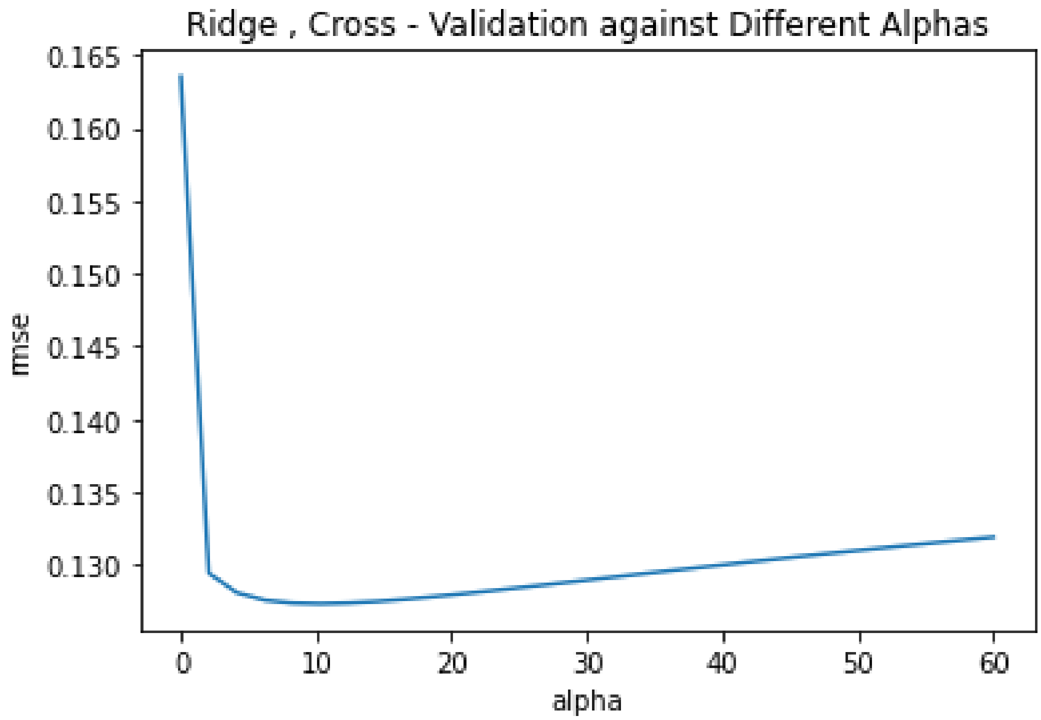
[ ]:
```
#fit the model. Usually now you use the best alpha determined by CrossValidation
model_ridge = Ridge(alpha = 0.1).fit(X_train,y)

#predict
ridge_preds = np.expm1(model_ridge.predict(X_test))

#Create file with predictions
solution = pd.DataFrame({"id":test.Id, "SalePrice":ridge_preds})
solution.to_csv("./predictions/ridge_sol.csv", index = False)
```

3. Compare a ridge regression and a lasso regression model. Optimize the alphas using cross validation. What is the best score you can get from a single ridge regression model and from a single lasso model?

[ ]:
```
alphas_ridge = np.linspace(1e-5, 60, 30)
cv_ridge = [rmse_cv(Ridge(alpha = alpha)).mean() for alpha in alphas_ridge]
```

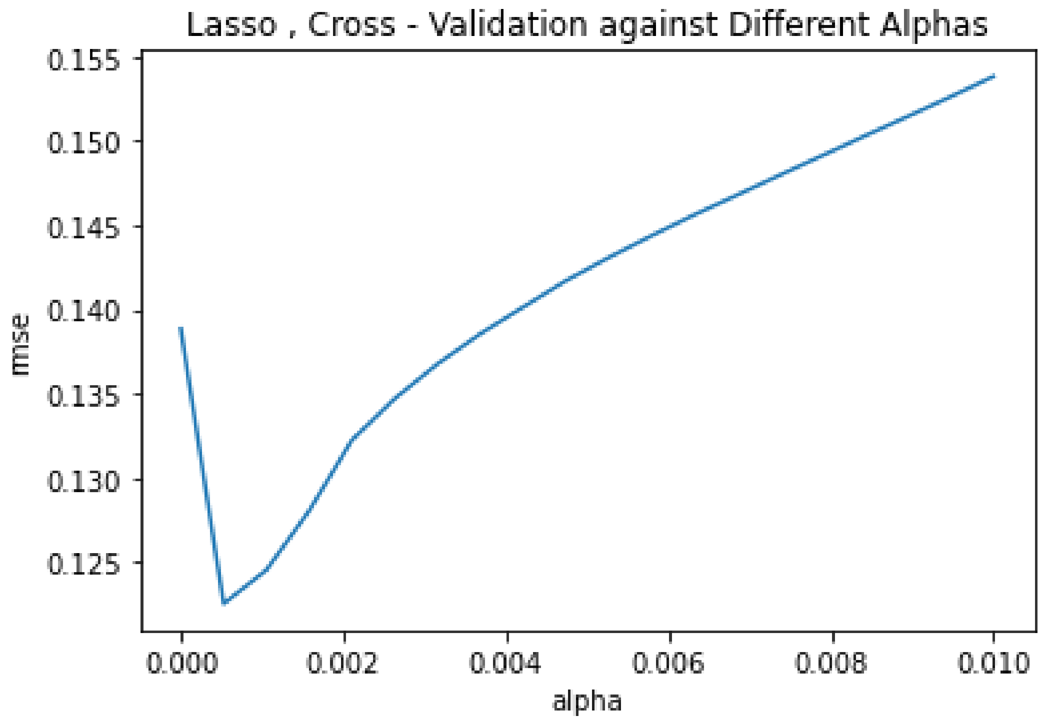
[ ]:
```
cv_ridge = pd.Series(cv_ridge, index = alphas_ridge)
cv_ridge.plot(title = "Ridge , Cross - Validation against Different Alphas")
plt.xlabel("alpha")
plt.ylabel("rmse")
```

[ ]:
```
Text(0, 0.5, 'rmse')
```




```
[ ]: alphas_lasso = np.linspace(1e-6, 1e-2, 20)
     cv_lasso = [rmse_cv(Lasso(alpha=alpha, max_iter=3000)).mean()for alpha in␣
      ↪alphas_lasso]

     cv_lasso = pd.Series(cv_lasso, index = alphas_lasso)
     cv_lasso.plot(title = "Lasso , Cross - Validation against Different Alphas")
     plt.xlabel("alpha")
     plt.ylabel("rmse")
```

```
[ ]: Text(0, 0.5, 'rmse')
```

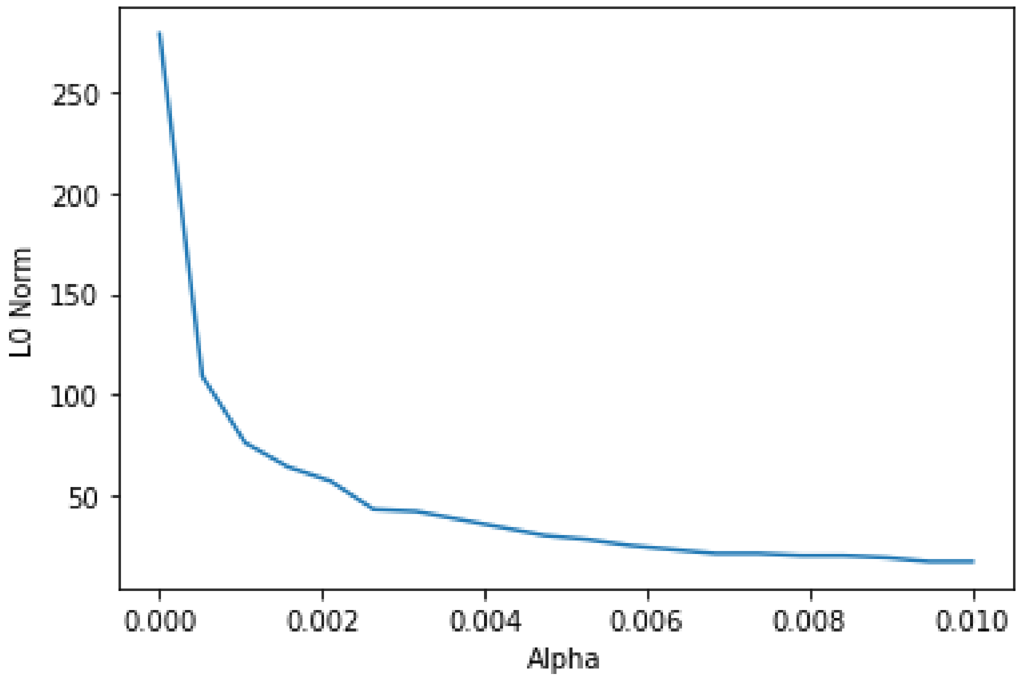


[ ]:
```
best_ridge_alpha =  alphas_ridge[np.argmin(cv_ridge)]
best_lasso_alpha = alphas_lasso[np.argmin(cv_lasso)]

print('Best Ridge RMSE {} for alpha {}'.format(cv_ridge.min(),␣
 ↪best_ridge_alpha))
print('Best Lasso RMSE {} for alpha {}'.format(cv_lasso.min(),␣
 ↪best_lasso_alpha))
```

```
Best Ridge RMSE 0.1273378832766949 for alpha 10.344835862068965
Best Lasso RMSE 0.12254501278784488 for alpha 0.0005272631578947369
```

4. Plot the l0 norm (number of nonzeros) of the coefficients that lasso produces as you vary the strength of regularization parameter alpha.

[ ]:
```
coef = [Lasso(alpha).fit(X_train,y).coef_ for alpha in alphas_lasso]
l0 = list()
for array in coef:
    l0.append(sum(array != 0))
plt.plot(alphas_lasso,l0)
plt.xlabel('Alpha')
plt.ylabel('L0 Norm')
plt.show()
```



5. Add the outputs of your models as features and train a ridge regression on all the features plus the model outputs (This is called Ensembling and Stacking). Be careful not to overfit. What score can you get? (We will be discussing ensembling more, later in the class, but you can start playing with it now).

We will now stack the models by feeding the predictions of our first two models into another model. If you use the data point label to train on the model, then there is some leakage of the labels when training the second model.

We split the data set into 5 folds. For each fold, we make a prediction on the data in this fold using a training set made up of the other 4 folds.

[ ]:
```
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
X_train_mat = X_train.values
kf = KFold(5)
n, d = X_train_mat.shape
X_stacked = np.concatenate([X_train_mat, np.zeros([n, 2])], axis=1)
ridge_rmse = []
lasso_rmse = []
for split_idx, val_idx in kf.split(X_train_mat, y):
    X_split = X_train_mat[split_idx]
    y_split = y[split_idx]
    X_val = X_train_mat[val_idx]
    y_val = y[val_idx]
```

```
    ridge = Ridge(best_ridge_alpha)
    ridge.fit(X_split, y_split)
    ridge_pred = ridge.predict(X_val)
    X_stacked[val_idx, -2] = ridge_pred
    ridge_rmse.append(np.sqrt(mean_squared_error(y_val, ridge_pred)))

    lasso = Lasso(best_lasso_alpha)
    lasso.fit(X_split, y_split)
    lasso_pred = lasso.predict(X_val)
    lasso_rmse.append(np.sqrt(mean_squared_error(y_val, lasso_pred)))
    X_stacked[val_idx, -1] = lasso_pred
print('Ridge RMSE: {}'.format(np.mean(ridge_rmse)))
print('Lasso RMSE: {}'.format(np.mean(lasso_rmse)))

ridge_2_cv= [(np.sqrt(-cross_val_score(Ridge(alpha), X_stacked, y,␣
 ↪scoring="neg_mean_squared_error", cv = 5))).mean() for alpha in alphas_ridge]
print('Best Stacked RMSE: {}'.format(np.min(ridge_2_cv)))
best_second_ridge_alpha = alphas_ridge[np.argmin(ridge_2_cv)]
```

```
Ridge RMSE: 0.12733788327669476
Lasso RMSE: 0.12254501278784483
Best Stacked RMSE: 0.10699734006125014
```

We see that stacking decreases the error. The results can be further enhanced by tuning the alphas of each of the models using cross validation. Now we make predictions on the test set:

[ ]:
```
ridge_1 = Ridge(best_ridge_alpha).fit(X_train, y)
lasso_1 = Lasso(best_lasso_alpha).fit(X_train, y)

ridge_2 = Ridge(best_second_ridge_alpha).fit(X_stacked, y)

X_test_aug = X_test.copy()
X_test_aug = np.concatenate([X_test_aug, np.zeros([X_test_aug.shape[0], 2])],␣
 ↪axis=1)
X_test_aug[:,-2] = ridge_1.predict(X_test)
X_test_aug[:,-1] = lasso_1.predict(X_test)

y_pred = np.expm1(ridge_2.predict(X_test_aug))

# Create file with predictions
solution = pd.DataFrame({"id":test.Id, "SalePrice":y_pred})
solution.to_csv("./predictions/aug_ridge_sol.csv", index = False)
```

6. Install XGBoost (Gradient Boosting) and train a gradient boosting regression.

[ ]:
```
import xgboost as xgb
```

```python
print(rmse_cv(xgb.XGBRegressor(n_estimators=300, learning_rate=0.1,␣
 ↪max_depth=2, objective='reg:squarederror')).mean())

xgb_model = xgb.XGBRegressor(n_estimators=300, learning_rate=0.1, max_depth=2,␣
 ↪objective='reg:squarederror')
xgb_model.fit(X_train, y)
xgb_preds = np.expm1(xgb_model.predict(X_test))

# Create file with predictions
solution = pd.DataFrame({"id":test.Id, "SalePrice":xgb_preds})
solution.to_csv("xgb_preds.csv", index = False)
```

```
0.12552195428748444
```

Results can be improved using cross validation over the parameters of XGBRegressor, as before.

7. For the final part of the exercise, there are several different ways to approach it. A simple extension of the above is to also include an XGBRegressor in the stacking of the models, and trying Lasso instead of Ridge for the second level, since it appears to perform better in the previous cases.

[ ]:
```python
X_train_mat = X_train.values
kf = KFold(5)
n, d = X_train_mat.shape
X_stacked = np.concatenate([X_train_mat, np.zeros([n, 3])], axis=1)

for split_idx, val_idx in kf.split(X_train_mat, y):
    X_split = X_train_mat[split_idx]
    y_split = y[split_idx]
    X_val = X_train_mat[val_idx]
    y_val = y[val_idx]

    ridge = Ridge(best_ridge_alpha)
    ridge.fit(X_split, y_split)
    ridge_pred = ridge.predict(X_val)
    X_stacked[val_idx, -3] = ridge_pred

    lasso = Lasso(best_lasso_alpha)
    lasso.fit(X_split, y_split)
    lasso_pred = lasso.predict(X_val)
    X_stacked[val_idx, -2] = lasso_pred

    xgb_model = xgb.XGBRegressor(n_estimators=300, learning_rate=0.1,␣
 ↪max_depth=2, objective='reg:squarederror')
    xgb_model.fit(X_split, y_split)
    xgb_pred = xgb_model.predict(X_val)
    X_stacked[val_idx, -1] = xgb_pred
```

```
final_lasso = [(np.sqrt(-cross_val_score(Lasso(alpha), X_stacked, y,␣
 ↪scoring="neg_mean_squared_error", cv = 5))).mean() for alpha in alphas_lasso]
print('Best Stacked RMSE: {}'.format(np.min(final_lasso)))
```

```
Best Stacked RMSE: 0.0959589371047462
```

Note that there is a slight improvement in the cross validation score, over the previous stacking implementation. This is due to the fact that the XGBoost model contributes to the ensemble of the first level predictors.

Note that the above is only one possible way to approach this part, so other extensions and implementations might lead to different, and possibly better, scores.