MY474 - Problem Set 5

29 March 2023

This is an individual assignment.

Exercise 1: Engineering missing value indicator features

In certain setups with tabular data that are commonly encountered in research and business cases, you will find that engineered features can potentially improve performance in prediction much more than the choice of the algorithm itself. In these cases, data scientists therefore often spend a lot of time carefully thinking about the set of features. This and the following exercise will study the topic in more detail.

The exemplary dataset we are going to look at is a sample of housing data from California. One line/observation in this dataset is a block. The task is to build a model that predicts the variable `median_house_value` as well as possible. You can find the data in the files `housing_train.csv` and `housing_test.csv`.

Loading packages:

```
library("tidyverse")
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.8
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.1     v forcats 0.5.1

## Warning: package 'dplyr' was built under R version 4.0.5

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library("glmnet")
```

```
## Loading required package: Matrix

##
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack

## Loaded glmnet 4.1-3
```

Loading the data:

```
train_data <- read_csv("data/housing_train.csv")
```

```
## Rows: 3000 Columns: 10-- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): ocean_proximity
## dbl (9): longitude, latitude, housing_median_age, total_rooms, total_bedroom...
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
test_data <- read_csv("data/housing_test.csv")
```

```
## Rows: 2000 Columns: 10-- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): ocean_proximity
## dbl (9): longitude, latitude, housing_median_age, total_rooms, total_bedroom...
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

In our example here, some features in the training and test datasets have missing values. In research and work as a data scientist this is a common case. One option is to drop all rows with one or more NA values, however, this also gets rid of a lot of information from the other columns (which can be particularly problematic if the dataset is small to begin with). In this exercise we are trying to use all information in the data and deal with missing values well.

There are many approaches to impute missing values, with the simplest being to choose the column mean (continuous variable) or mode (categorical variable). More important than the imputation technique itself (!), however, is often to signal to the model that a value has been imputed at all. This is what we are going to study.

a) Impute missing values in the continuous features of `train_X` and `test_X` with their column mean from `train_X`. This assumes that new test data will come in at high frequency in later implementations of the model, and that it will not be possible to recompute the means including also the test data every time. For missing values in the categorical variable, replace them with the mode of the variable in the training data. Next, using the training data, cross validate a LASSO (note: using the function `lasso_model <- cv.glmnet(...)` does both in one - it chooses the best lambda via cross validation and can also be used to predict afterwards). What is the RMSE that this model achieves on the test data?

To answer this question, read through the code to make sure to understand it and just fill in the blanks: (1.5 points)

```
# Helper function for getting the mode in R (source: https://stackoverflow.com/questions/2547402/how-to
get_mode <- function(x) {
  ux <- unique(x)
  ux[which.max(tabulate(match(x, ux)))]
}

# Because the following imputation is only a reference, we will wrap the code
# into a function such that we can run it and see the associated RMSE but avoid
# having many additional objects stored in the global environment afterwards
simple_imputation <- function() {
```

```r
  # Copies of raw data
  train <- train_data
  test <- test_data

  # Computing means using only training data
  means <- train %>% select(population, median_income) %>% colMeans(na.rm = TRUE)

  # Imputing missing values of continuous features with train means
  train[is.na(train$population), "population"] <- means["population"]
  train[is.na(train$median_income), "median_income"] <- means["median_income"]
  test[is.na(test$population), "population"] <- means["population"]
  test[is.na(test$median_income), "median_income"] <- means["median_income"]

  # Adding the mode for missing categorical variable values
  train[is.na(train$ocean_proximity), "ocean_proximity"] <- get_mode(train$ocean_proximity)
  test[is.na(test$ocean_proximity), "ocean_proximity"] <- get_mode(train$ocean_proximity)

  # training and test X/y
  train_X <- model.matrix(~., train %>% select(-median_house_value))
  test_X <- model.matrix(~., test %>% select(-median_house_value))

  # Why do we use `model.matrix` to compute the input for the glmnet's LASSO?
  # It transforms categorical levels into 0/1 and and also adds an intercept column

  # Train/test y's
  train_y <- train %>% pull(median_house_value)
  test_y <- test %>% pull(median_house_value)

  # Train lasso model
  lasso_model <- cv.glmnet(x = train_X, y = train_y)

  # Predict train_y_hat and test_y_hat
  train_y_hat <- predict(lasso_model, train_X)
  test_y_hat <- predict(lasso_model, test_X)

  # Test RMSE
  print(sqrt(mean((test_y - test_y_hat)^2)))

}

# Running simple imputation
simple_imputation()
```

```
## [1] 85884.77
```

b) The approach above is a naive way to deal with missing values. Our first engineered features will therefore be new columns with missing value indicators. Add one indicator column to both `train_X` and `test_X` for each *continuous* variable with missing values. These indicator columns take a value of 1 if the value in a corresponding variable was imputed and zero otherwise. Conveniently, for each categorical variable, you can just add a new label if an observation is missing (instead of using the mode). Train a model with the new `train_X` matrix containing the indicators and predict with the new `test_X` matrix containing the indicators.

To answer this question, again fill in the blanks in the following code. What test RMSE do you find now? Lastly, when do you think will adding such indicator columns be most important? (1.5 points)

```
# Copies of raw data
train <- train_data
test <- test_data

# Computing means using only training data
means <- train %>% select(population, median_income) %>% colMeans(na.rm = TRUE)

# Adding missing value indicators for continuous features
train$population_indicator <- as.numeric(is.na(train$population))
train$median_income_indicator <- as.numeric(is.na(train$median_income))
test$population_indicator <- as.numeric(is.na(test$population))
test$median_income_indicator <- as.numeric(is.na(test$median_income))


# Imputing missing values of continuous features with train means
train[is.na(train$population), "population"] <- means["population"]
train[is.na(train$median_income), "median_income"] <- means["median_income"]
test[is.na(test$population), "population"] <- means["population"]
test[is.na(test$median_income), "median_income"] <- means["median_income"]

# Adding new labels for categorical variables
train[is.na(train$ocean_proximity), "ocean_proximity"] <- "not_available"
test[is.na(test$ocean_proximity), "ocean_proximity"] <- "not_available"

# training and test X/y
train_X <- model.matrix(~., train %>% select(-median_house_value))
test_X <- model.matrix(~., test %>% select(-median_house_value))
# Why do we use `model.matrix` to compute the input for the glmnet's LASSO?
# What does `model.matrix` do here?

# Train/test y's
train_y <- train %>% pull(median_house_value)
test_y <- test %>% pull(median_house_value)

# Train lasso model
set.seed(123)
lasso_model <- cv.glmnet(x = train_X, y = train_y)

# Predict train_y_hat and test_y_hat
train_y_hat <- predict(lasso_model, train_X)
test_y_hat <- predict(lasso_model, test_X)

# Test RMSE
print(sqrt(mean((test_y - test_y_hat)^2)))
```

```
## [1] 68402.31
```

Exercise 2: General feature engineering

Unfortunately there is no standardised approach to engineering features that always manages to improve predictions. In the following, I have therefore assembled a list of some ideas and thoughts on feature

engineering. Many of these might eventually not improve a particular problem, but if you have the time it can be worth it to try these and similar approaches. It is often the combination of many small improvements which determine the best model. In business applications, small improvements in predictions can also translate into relevant financial gains (think e.g. of having slightly better predictions of prices which consumers would be willing to pay in a large web shop). Only very few of the bullet points listed below might help/be applicable for this particular problem, the purpose of the list is also to be a starting point which might be helpful for your future work.

- Is it possible to gather data on new features which could have predictive power for the respective problem? It can be worth it to first take a step back and think about what the ideal predictive feature for a particular problem could be. Would it be possible to obtain data for this feature or a similar one? You can also use measures of feature importance to get a broad overview which group of features is more predictive and whether you can find similar new data, but there is clear value in first thinking about the problem more abstractly before running such computations.
- Whether you impute missing values in continuous features just with the mean or with an advanced method, it is key to add new indicator features as we have done in Exercise 1.
- Create histograms of features, do some have very skewed distributions with outliers which might throw off predictions? Potentially try transformations such as the log for these features. Do you find that the transformed features look more normally distributed in histograms? Then it might be useful to use those. Note, however, that when you try to predict y values with a lot of outliers and only use a linear model, the outliers in the x features before any log transform can also be helpful in predicting the outlier y values. In contrast, a non-linear model could also predict y outliers well if the corresponding x values have a more normally distributed shape and indeed further benefit from less erratic feature values.
- Try to winsorize or remove extreme outliers in the outcome variable or features that might bias predictions on the non-outlier observations. Another approach is to log transform the outcome variable. Beware though that by Jensen's inequality your predictions will be biased if you transform the outcome variable back with exp(y_hat) afterwards. This bias might be small, however, for a given application
- To minimise the negative impact of outlier observations in the outcome variable on the non-outlier observation without winsorizing or removing outliers, it can also be useful to use a Huber loss function in regression. Flexible models like LightGBM allow to swap the loss function relatively easily.
- It can be interesting to check pairwise correlations of features with the outcome variable. Are there some clear associations, potentially non-linear? Then it might be helpful to either try adding higher order terms of the feature or even discretising it into several indicator variables that change their value to 1 if the feature is in a specific range (this allows the model to fit very non-linear relations between the feature and the outcome variable).
- Another approach is more agnostic, namely to add a range of transformations to the data e.g. squares and interactions (try e.g. `model.matrix( ~.^n, data = some_data)` which creates interactions of order `n` or the function step_interact). After you have blown up the number of columns in feature matrix, train e.g. a LASSO or other method that can deal well with many columns. Caveat: Recall that the combinatorics implied by even a very small number `n` can create a very large amount of new columns which can become difficult to handle for the computer's memory and greatly increase training time. Furthermore, higher order terms can behave quite erratically, so it is important to thoroughly test such models using the new transformation on out of sample data.
- When having high-dimensional categoricals, it is usually the case that many of the levels have very few observations. To make training faster and also avoid over-fitting on levels with very few observations, it can be worth it to pool rare levels into an "other" category. This also makes it easier when encountering new observations that have an unknown but rare level in this column and which can then just be assigned the value "other" when using it in a prediction.

- Do you have alternative data sources such as texts, e.g. when predicting the price of a good with an additional product description? You can either apply domain knowledge and extract terms with regular expressions that are relevant to a specific problem (e.g. brands in descriptions of clothes, "gmail" vs "hotmail" email addresses signalling different types of customers, etc.) or take a more agnostic approach and create a document term matrix from the text, normalise it by document length, and add it to your features (yet this would make the problem very high dimensional as every unique word would become an additional feature). Furthermore, when working with texts, also other representations of terms can be helpful such as (pre-trained) word embeddings which can be freely downloaded for most words or context specific word embeddings obtained with transformers.

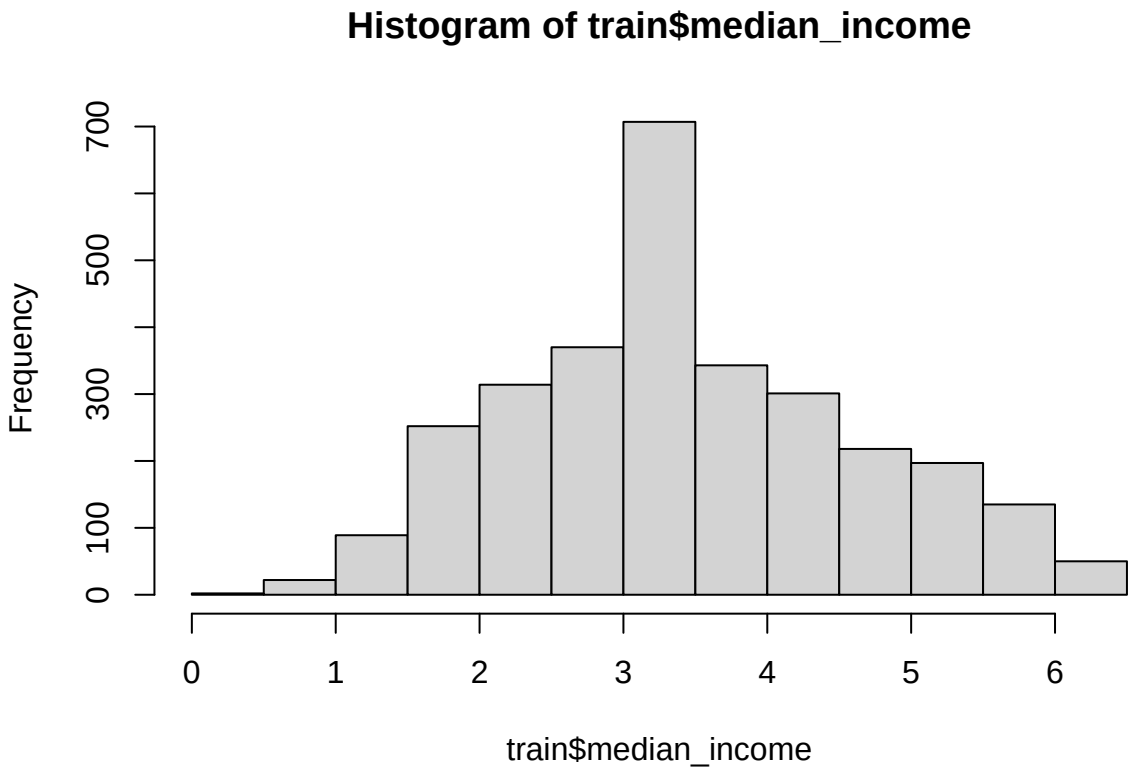
Using your **LASSO model (it is not allowed in this exercise to user other algorithms)** built previously including the missing value indicators, modify the code and present a few potential transformations etc. to engineer features for this dataset. Why did you try these in particular?

Could you further bring down the test RMSE predicting `median_house_value` now also using further engineered features? If you could not reduce the test RMSE further from relative to Exercise 1 b), what could be interesting data to collect here? Please clearly report the best test RMSE value you could achieve. (5 points)

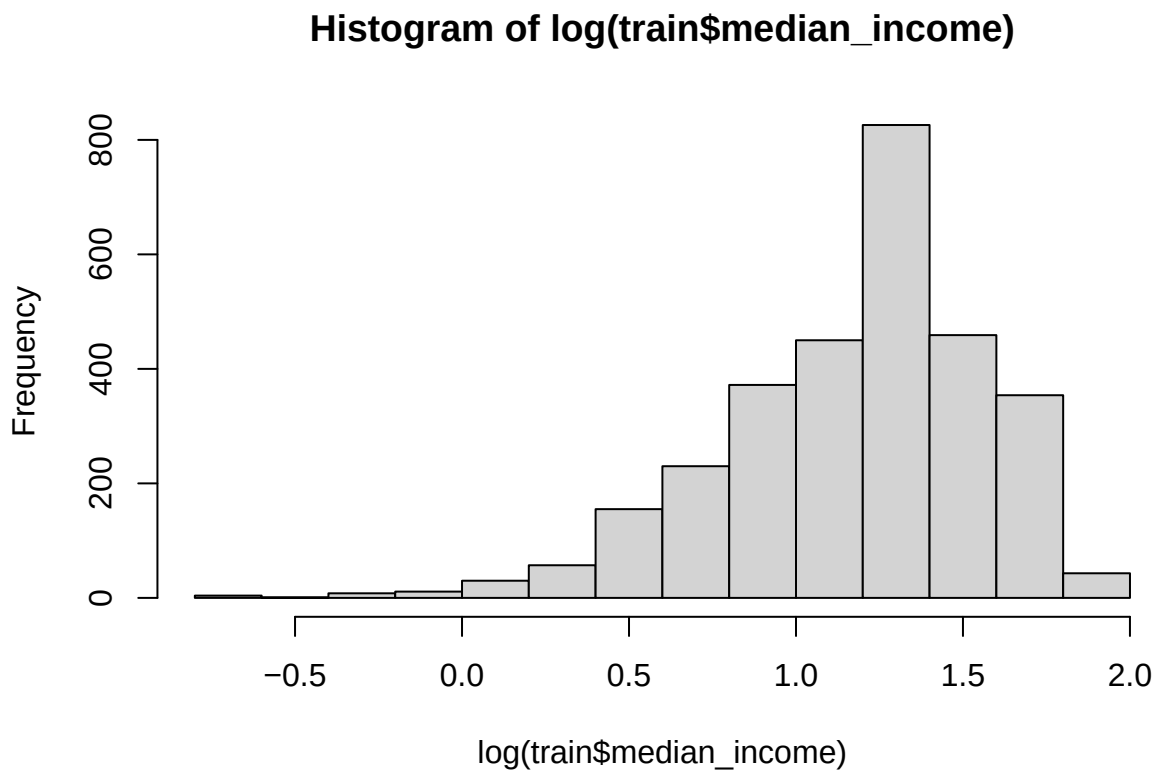
Histograms are a useful way to visualize the distribution of each feature in the dataset. A skewed distribution may indicate the presence of outliers or extreme values, which can impact the performance of predictive models. In such cases, a log transformation can be applied to the feature to make it more normally distributed.

let us take one feature as na example

```
hist(train$median_income)
```



```
hist(log(train$median_income))
```



we can see that logging does not really help, so we will not use it

```
library(geosphere)

# Copies of raw data
train <- na.omit(train_data)
test <- na.omit(test_data)

# Computing means using only training data
means <- train %>% select(population, median_income) %>% colMeans(na.rm = TRUE)

# Adding missing value indicators for continuous features
train$population_indicator <- as.numeric(is.na(train$population))
train$median_income_indicator <- as.numeric(is.na(train$median_income))
test$population_indicator <- as.numeric(is.na(test$population))
test$median_income_indicator <- as.numeric(is.na(test$median_income))

# Adding interaction terms

train$ocean_proximity[train$ocean_proximity=="NEAR BAY"] <- 1
train$ocean_proximity[train$ocean_proximity=="<1H OCEAN"] <- 2
train$ocean_proximity[train$ocean_proximity=="NEAR OCEAN"] <- 3
train$ocean_proximity[train$ocean_proximity=="ISLAND"] <- 4
train$ocean_proximity[train$ocean_proximity=="INLAND"] <- 5
```

```r
train$ocean_proximity[train$ocean_proximity=="not_available"] <- 0


test$ocean_proximity[test$ocean_proximity=="NEAR BAY"] <- 1
test$ocean_proximity[test$ocean_proximity=="<1H OCEAN"] <- 2
test$ocean_proximity[test$ocean_proximity=="NEAR OCEAN"] <- 3
test$ocean_proximity[test$ocean_proximity=="ISLAND"] <- 4
test$ocean_proximity[test$ocean_proximity=="INLAND"] <- 5
test$ocean_proximity[test$ocean_proximity=="not_available"] <- 0

train$ocean_proximity <- as.numeric(train$ocean_proximity)
test$ocean_proximity <- as.numeric(test$ocean_proximity)

train$median_income_ocean_proximity <- train$median_income * train$ocean_proximity
test$median_income_ocean_proximity <- test$median_income * test$ocean_proximity
train$population_housing_median_age <- train$population * train$housing_median_age
test$population_housing_median_age <- test$population * test$housing_median_age



# Adding bedroom/room ratio feature
train$bedroom_room_ratio <- train$total_bedrooms / train$total_rooms
test$bedroom_room_ratio <- test$total_bedrooms / test$total_rooms

# Imputing missing values of continuous features with train means
train[is.na(train$population), "population"] <- means["population"]
train[is.na(train$median_income), "median_income"] <- means["median_income"]
test[is.na(test$population), "population"] <- means["population"]
test[is.na(test$median_income), "median_income"] <- means["median_income"]



# training and test X/y
train_X <- model.matrix(~., train %>% select(-median_house_value))
test_X <- model.matrix(~., test %>% select(-median_house_value))


# Train/test y's
train_y <- train %>% pull(median_house_value)
test_y <- test %>% pull(median_house_value)

# Train lasso model
set.seed(123)
lasso_model <- cv.glmnet(x = train_X, y = train_y)

# Predict train_y_hat and test_y_hat
train_y_hat <- predict(lasso_model, train_X)
test_y_hat <- predict(lasso_model, test_X)

# Test RMSE
print(sqrt(mean((test_y - test_y_hat)^2)))
```

```
## [1] 64036.79
```

Exercise 3

Much discussion about machine learning and AI over the recent months has been driven by large language models (LLMs). At the moment, these models are typically decoder transformers of the kind that we discussed in the lecture. For now the GPT models such as ChatGPT are most popularised and able to follow instructions quite well, but there are several models from other providers available and more will be released in the upcoming months and years.

There likely exist a multitude of potential research projects in the social sciences where such models could be utilised. For some social science research involving texts, you can e.g. think of an LLM like an electronic research assistant that can process texts very well and at scale, summarise them, extract information from them, etc. You could repeatedly submit short to medium length texts to the model and ask questions about these. For example, a query to the model might look like:

"[Some excerpt from an economic text pasted here, e.g. from a newspaper or company report]

Yes or no, does this text deal with environmental issues? If yes, summarise the environmental issues in less than 200 words. Lastly, is the sentiment of the text likely positive, negative, or neutral?"

The response text generated by the model will contain a sentiment label and potentially also a summary. These pieces of information could be separated and stored in a row of a tabular dataset (e.g. to then analyse subsequently with simpler methods). Afterwards the next newspaper text/company report could be parsed into the model with the same question added below it, and so on. In the end, a dataset assembled from such prompts might e.g. be used in a wider research paper analysing environmental discussions in company reports and how they changed over time.

This exercise is meant as a space for you to brainstorm and develop ideas how LLMs could be used in a research project in your field of the social sciences. Write a short research proposal/extended abstract of **up to 600 words** where you try to develop a research idea. This text is **not meant to be only generic** like "One could use LLMs in financial economics research", but instead meant to describe a proposed research idea/project as precisely as possible given the word limit. Make sure to cite relevant literature and sources where appropriate and list them in the references. While developing the idea, you might further read into large language models. One important particularity of these models to recall from the lecture is their maximum context window length – it is so far not possible to parse very long texts into them in a single query and have them understand all text at once. So a research idea containing "I paste the following book that contains 2000 pages into the model as only a single combined input text and ask it to summarise this text" would not be feasible. Reading up on these and other limitations will make sure that your short research proposal remains realistic in terms of what these models currently can and also cannot do.

Begin the answer by stating a research question. As part of your answer, you can also e.g. mention potential prompts, how their answers by the LLM could be used for assembling a dataset for the project, short details of the subsequent analysis, for which tasks simpler approaches also covered in this course might be preferable (e.g. for plain sentiment analysis, a simpler tool might be almost as good, but much faster and cheaper then sending each text to an LLM), or which current limitations of LLMs could be relevant for the research project. Lastly, try to develop a new idea in this exercise rather than one related to another project you are/have been working on, but if it relates to any other of your work, make sure to cite it clearly in the references. (7 points)

More original and concrete answers that display more detailed understanding of methods discussed in this course will receive more points here. Also see the marking criteria at https://lse-my474.github.io/ for further information.

Research Question: Can large language models be used to analyse public opinion on vaccination in social media and news articles?

Background and Motivation: Vaccination is one of the most effective ways to prevent the spread of infectious diseases. However, vaccine hesitancy and anti-vaccination movements have become increasingly popular in recent years. Social media platforms such as Twitter and Facebook have played a key role in disseminating vaccine-related information and shaping public opinion on vaccination. News articles and

online forums are also important sources of vaccine-related information. There is a need for an automated tool that can analyze public opinion on vaccination in real-time, monitor trends and identify influential sources of information.

Methodology: Large language models such as GPT-3 can be trained on a vast amount of text data to perform a variety of natural language processing tasks, including sentiment analysis, summarization, and named entity recognition. The model can be used to analyze vaccine-related tweets, news articles, and online forum posts, and extract information such as sentiment, key topics, and influential sources. The model can be trained on a large dataset of labeled tweets and articles to improve its performance.

Data Collection: Twitter, news articles, and online forums can be scraped using APIs provided by the respective platforms. The data can be filtered using keywords related to vaccination. To ensure the quality of the data, a random sample of tweets and articles can be manually annotated for sentiment, relevance, and accuracy. The labeled data can be used to train the model and evaluate its performance.

Data Analysis: The sentiment of vaccine-related tweets and news articles can be analyzed using the model. The model can be used to summarize the main topics discussed in the tweets and articles and identify influential sources of information. The output of the model can be visualized using dashboards that allow real-time monitoring of public opinion on vaccination. In addition, the model can be used to identify misinformation and conspiracy theories related to vaccination.

Limitations: The model's performance is dependent on the quality of the labeled data used to train it. The model may not perform well on texts that are significantly different from the training data. The model may also struggle with sarcasm and irony, which are common in social media. Finally, the model's predictions may not always be accurate, and manual verification may be required.

Conclusion: Large language models can be used to analyze public opinion on vaccination in social media and news articles. The model can provide real-time monitoring of trends and identify influential sources of information. However, the model's predictions may not always be accurate, and manual verification may be required.

References:

Mavragani, A., Ochoa, G., & Tsagarakis, K. P. (2019). Assessing the methods, tools, and statistical approaches in Google Trends research: systematic review. Journal of medical Internet research, 21(11), e13473. Garg, S., Williams, N. A., & Ip, A. (2020). The digital revolution: Using social media to assess and promote vaccine hesitancy and acceptance. Vaccine, 38(39), 6257-6259. Jain, V., Sharma, M., & Purohit, H. (2021). Sentiment Analysis of COVID-19-Related Tweets: A Comprehensive Review. IEEE Transactions on Computational Social Systems, 8(1), 1-15.

Note: For a bit of background how one would actually send texts to such a model in a research project – For now, many LLMs cannot be run locally on computers because they are too large, but instead are commonly used via browsers. In a research project, however, it would be much easier to query the underlying model via an API (Application Programming Interface; if you are interested in the topic, have a look at the recording of week 8 in MY472) which are also often offered by the providers. In a nutshell, querying the model via an API means to send the input text with a programming language such as R or Python over the internet and receive the answer of the model (e.g. the response text or a numerical embedding of the text – the LLMs can also return only embeddings that could be used for own downstream tasks!) back directly in the programming language. This allows to send many smaller texts to the model via loops and to process their returns directly in the programming language rather than using a visual user interface in a browser.

Exercise 4

Use the cifar data set from the lecture coding example `02-cnn.Rmd`. Keeping the training and test samples exactly the same as in the lecture code, can you achieve a higher accuracy than what we found in the lecture? The purpose of this exercise is to independently study some interesting concepts in neural networks which can increase predictive performance. You might e.g. want to look into topics such as batch normalisation, dropout (note: it is generally not advisable to use dropout in the convolutional layers), or other forms of regularisation. Furthermore, you can try different combinations of layers, amounts of filter, the kernel

sizes of the individual filters, different numbers of training episodes (while the training loss decreases, did the validation loss already increase?), etc. Another approach is to explore alternative model architectures altogether which might improve the performance for the cifar task as well. You can find many alternative models here: https://github.com/rstudio/keras/tree/master/vignettes/examples.

Which **test set** accuracy does your best model achieve? Which changes could improve the model? (5 points)

Alternative Exercise 4

Only students on whose computers Keras does not work in R may attempt this alternative exercise. If this is the case for you, please make sure to take screenshots of the precise error messages in your case and submit the screenshots in the assignment folder as well.

Using and tuning any algorithm discussed in this course and training it on `housing_train.csv`, what is the very best RMSE you can achieve predicting `median_house_value` on the test data `housing_test.csv`? You can also use algorithms/R packages not discussed in the course as long as you can describe how they work (Part 4.2). Feel free to keep your engineered features from Exercise 1 if they turn out helpful with the new model as well, but the main task here is to explore how much models other than the LASSO used previously can improve predictions.

What is the lowest test RMSE you can achieve with your best prediction model? Which algorithms could improve the prediction? (5 points)

To answer this question, I will train a Random Forest model on the housing_train.csv data and tune its hyperparameters using cross-validation.

```
library(caret)
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
# read in the data
train <- na.omit(read.csv("data/housing_train.csv"))
test <- na.omit(read.csv("data/housing_test.csv"))

# fit random forest model
rf_model <- randomForest(median_house_value ~ ., data = train, importance = TRUE, ntree = 1000, nodesiz

# predict on test set
rf_pred <- predict(rf_model, test)

# calculate RMSE
rmse <- sqrt(mean((test$median_house_value - rf_pred)^2))
rmse
```

```
## [1] 52209.02
```

```r
# Set up the cross-validation method
ctrl <- trainControl(method = "cv", number = 5)

# Set up the parameter grid to tune over
param_grid <- expand.grid(
  n_estimators = c(50, 500, 1000, 1500),
  max_depth = c(5, 10, 15),
  min_samples_split = c(2, 5, 10),
  min_samples_leaf = c(1, 2, 4)
)



# Train the model with cross-validation
set.seed(123)

param_grid <- expand.grid(mtry = seq(2, 10, by = 2))

rf_fit <- train(median_house_value ~ ., data = train, method = "rf", trControl = ctrl, tuneGrid = param

rf_fit
```

```
## Random Forest
##
## 2214 samples
##    9 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 1772, 1772, 1770, 1771, 1771
## Resampling results across tuning parameters:
##
##   mtry  RMSE      Rsquared   MAE
##    2    56956.24  0.6719755  39580.80
##    4    54710.56  0.6838403  37370.87
##    6    54724.73  0.6804425  37152.27
##    8    55229.59  0.6726037  37474.71
```

```
##   10    55462.11  0.6691835  37608.87
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 4.
```

Go to file/function | Addins

Untitled.Rmd* | problem-set-5.Rmd | 02-cnn.Rmd* | feedback.Rmd

Knit | Run

```
11 Loading packages:
12
13 ```{r}
14 library("tensorflow")
15 library("keras")
16 library(reticulate)
17
18 cifar <- dataset_cifar10()
19 ```
```

Error in python_config_impl(python) : Error 1 occurred running /usr/bin/python3:

Hide Traceback

15. stop("Error ", status, " occurred running ", python, ": ", errmsg)
14. python_config_impl(python)
13. python_config(python_version, required_module, python_versions)
12. py_discover_config(required_module, use_environment)
11. (function() { op <- options(reticulate.python.initializing = TRUE) on.exit(options(op), add = TRUE) py_discover_config(required_module, use_environment) ...
10. initialize_python(required_module, use_environment)
9. ensure_python_initialized()
8. py_config()
7. tf_config()
6. stop(tf_config()$error_message, call. = FALSE)
5. on_error(result)
4. py_resolve_module_proxy(x)
3. `$.python.builtin.module`(keras, "datasets")
2. keras$datasets
1. dataset_cifar10()

```
20
```