mlp-week06

March 3, 2021

1 Machine Learning in Python - Workshop 6

2 1. Setup

2.1 1.1 Packages

In the cell below we will load the core libraries we will be using for this workshop and setting some sensible defaults for our plot size and resolution.

[1]:
```
# Display plots inline
%matplotlib inline

# Data libraries
import pandas as pd
import numpy as np

# Plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Plotting defaults
plt.rcParams['figure.figsize'] = (10,6)
plt.rcParams['figure.dpi'] = 80

# ipython interactive widgets
from ipywidgets import interact

# sklearn modules
import sklearn
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV, KFold
```

3 2. Kernel Ridge Regression

This is an approach that combines Ridge regression with the kernel trick to allow for the fitting of more complex relationships. Beginning with the Ridge regression optimization formulation,

$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \alpha(\boldsymbol{\beta}^T\boldsymbol{\beta})$$

We have seen that it has a closed form solution where

$$\boldsymbol{\beta} = \left(\boldsymbol{X}\boldsymbol{X}^\top + \alpha\boldsymbol{I}_p\right)^{-1}\boldsymbol{X}^\top\boldsymbol{y}$$

which can be rewritten as

$$\begin{aligned}\boldsymbol{\beta} &= \boldsymbol{X}^\top\left(\boldsymbol{X}^\top\boldsymbol{X} + \alpha\boldsymbol{I}_p\right)^{-1}\boldsymbol{y} \\ &= \boldsymbol{X}^\top\left(\boldsymbol{K} + \alpha\boldsymbol{I}_p\right)^{-1}\boldsymbol{y}\end{aligned}$$

where $\boldsymbol{K}$ is a Gram matrix defined by a given kernel.

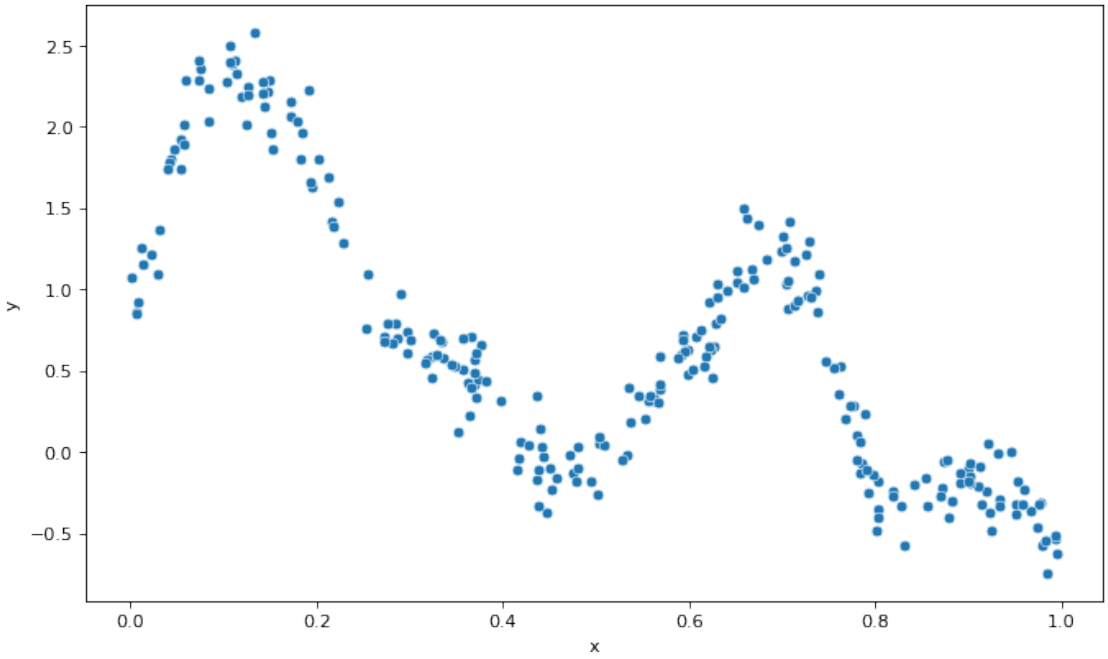
The implementation in scikit-learn is provided by the `KernelRidge` function in the `sklearn.kernel_ridge` submodule.

[2]:
```
from sklearn.kernel_ridge import KernelRidge
```

We will use this model to fit the gaussian process data we originally examined in week 3.

[3]:
```
gp = pd.read_csv('gp.csv').sample(frac=1)
```

[4]:
```
sns.scatterplot(x='x', y='y', data=gp)
plt.show()
```



[5]:
```
# Define outcome vector and model matrix
y = gp.y
X = gp.drop('y', axis=1)
```

3.1 2.1 Model Parameters

To fit a kernel ridge regression model we first must choose a kernel function, here we will be using a radial basis function (`rbf`), other possible choices are available as part of the `sklearn.metrics.pairwise` submodule. Each kernel will have additional parameters, generally sklearn attempts to label all primary kernel parameters using the name `gamma` and any additional required parameters will have more informative names. In the lectures this parameter was referred to as the bandwidth and labelled as l. The parameterization of the radial basis function used by scikit-learn is as follows:

$$K(\boldsymbol{x}, \boldsymbol{y}) = \exp(-\gamma \|\boldsymbol{x} - \boldsymbol{y}\|^2).$$

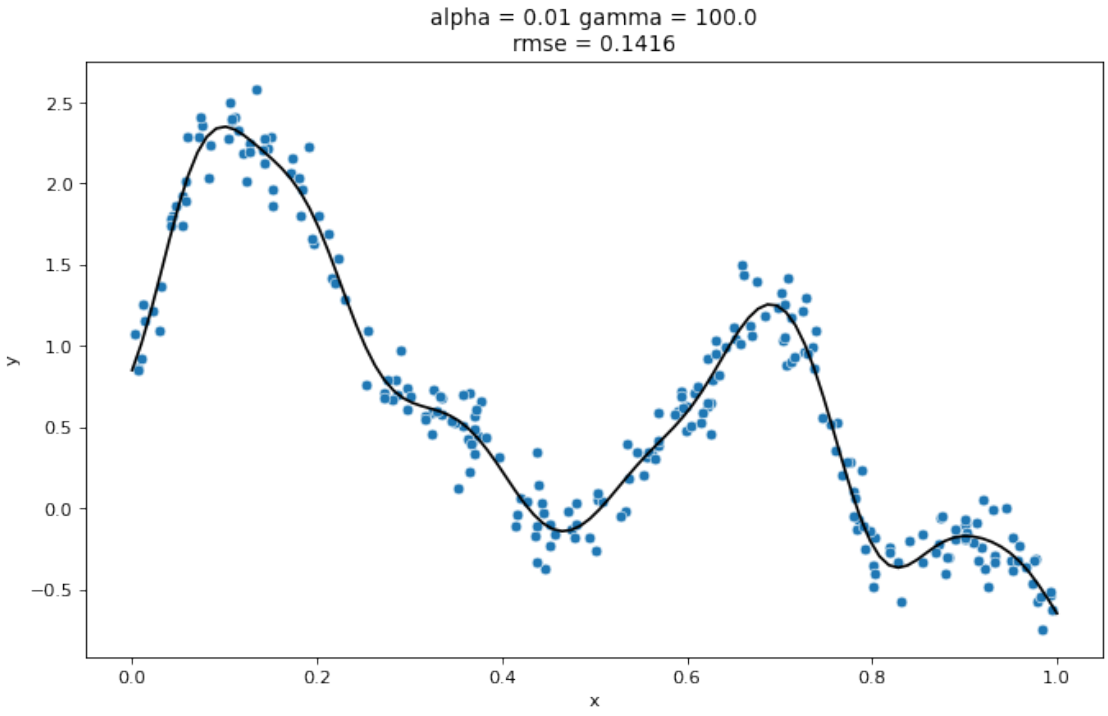
As such, when fitting a rbf based kernel ridge regression model we now have two parameters to optimize over: `alpha` from the ridge ℓ_2 penalty, and `gamma` the bandwidth of the rbf kernel.

Below we define a function that fits the kernel ridge regression to the `gp` data and then plots the fit along with reporting the root mean squared error. By including the `interact` decorator we provide simple interactive widgets that you can use to adjust these two parameters to explore their effect on the model's fit.

[6]:
```
def kernel_ridge_fit(alpha, gamma):
    # Fit the model
    m = KernelRidge(kernel='rbf', alpha=alpha, gamma=gamma).fit(X, y)

    rmse = np.sqrt(mean_squared_error(y, m.predict(X)))

    # Create DF for predictions
    gp_pred = pd.DataFrame({'x': np.linspace(0, 1, num=100)})
    gp_pred["y"] = m.predict(gp_pred)

    sns.scatterplot(x='x', y='y', data=gp)
    sns.lineplot(x='x', y='y', data=gp_pred, color='black')
    plt.title("alpha = %s gamma = %.1f\nrmse = %.4f" % (alpha, gamma, rmse))
    plt.show()
```

You can try adjusting the values of these parameters in the code chunk below and exploring the effect they have on the overall model fit. We recommend trying values of `alpha` between 0 and 1 and `gamma` between 0 and 1000. Note the behavior of `gamma` will depend on your choice of `alpha`.

[7]:
```
kernel_ridge_fit(alpha = 0.01, gamma = 100)
```



3.1.1 Exercise 1

How does changing the value of `alpha` and `gamma` affect the fit of this model? Which of the two seems more important? Explain.

A larger value of gamma(1000) leads to a lower rmse, however, we are overfitting the data in this case and choosing large values is not ideal.

A larger value of alpha will more harshly penalise larger coefficients and increasing it reduces the rmse.

Therefore picking a low value of alpha (0.01) and a modest value of gamma (100) that avoids overfitting results in a lower rmse, but I believe this model will perform better on unseen test data.

3.2 2.2 Model Tuning

As with ridge and lasso regression models we can use `GridSearchCV` to explore the space of possible parameters to determine the optimal model under cross validation.

[8]:
```
m = GridSearchCV(
    KernelRidge(kernel='rbf'),
    param_grid={"alpha": np.logspace(-3, 0, num=4),
                "gamma": np.logspace(0, 3, num=50)},
    scoring = 'neg_mean_squared_error',
    cv = KFold(5, shuffle=True, random_state=1234)
)

m = m.fit(X,y)
```

Note that with two parameters we now need to fit $n_\alpha \times n_\gamma \times n_{cv}$ models which can be somewhat slow.

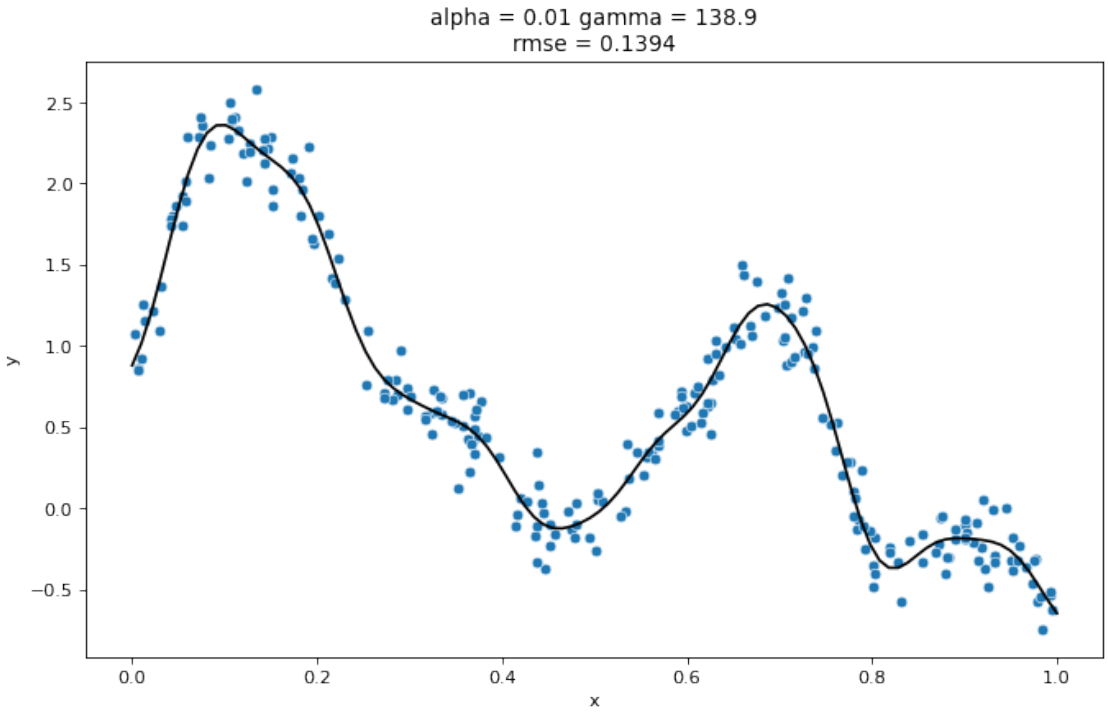
For our grid search we obtain the following as the best values of these parameters:

[9]:
```
m.best_params_
```

[9]:
```
{'alpha': 0.01, 'gamma': 138.94954943731375}
```

Once the parameter values have been determined we can then examine the model's performance with the entire data set using the `kernel_ridge_fit` function we previously defined.

[10]:
```
kernel_ridge_fit(m.best_params_["alpha"], m.best_params_["gamma"])
```



3.2.1 Exercise 2

Repeat the grid search above but fix `alpha` to be 0.01, 0.1 and 1 and search over the possible values of `gamma`. For each of these three models compare the rmses of the models.

[11]:
```
alpha = 0.01

m1 = GridSearchCV(
    KernelRidge(kernel='rbf'),
    param_grid={"alpha": np.array([alpha]),
                "gamma": np.logspace(0, 3, num=50)},
    scoring = 'neg_mean_squared_error',
    cv = KFold(5, shuffle=True, random_state=1234)
)

m1 = m1.fit(X,y)

m1.best_params_
```

[11]:
```
{'alpha': 0.01, 'gamma': 138.94954943731375}
```

[12]:
```
alpha = 0.1

m2 = GridSearchCV(
    KernelRidge(kernel='rbf'),
    param_grid={"alpha": np.array([alpha]),
                "gamma": np.logspace(0, 3, num=50)},
    scoring = 'neg_mean_squared_error',
    cv = KFold(5, shuffle=True, random_state=1234)
)

m2 = m2.fit(X,y)

m2.best_params_
```

[12]:
```
{'alpha': 0.1, 'gamma': 212.09508879201903}
```

[13]:
```
alpha = 1

m3 = GridSearchCV(
    KernelRidge(kernel='rbf'),
    param_grid={"alpha": np.array([alpha]),
                "gamma": np.logspace(0, 3, num=50)},
    scoring = 'neg_mean_squared_error',
    cv = KFold(5, shuffle=True, random_state=1234)
)

m3 = m3.fit(X,y)

m3.best_params_
```

[13]:
```
{'alpha': 1, 'gamma': 159.98587196060572}
```

[14]:
```
#print(np.logspace(-3,0,num=4))
#print(np.array())

print(np.sqrt(mean_squared_error(y, m1.predict(X))))
print(np.sqrt(mean_squared_error(y, m2.predict(X))))
print(np.sqrt(mean_squared_error(y, m3.predict(X))))
```

```
0.13942955544644967
0.13931254097003104
0.1493598531944324
```

Our best model used alpha = 0.1, so we expect that the rmse of m2 will be the smallest. We also expect that alpha = 1 will be the worst from trying different values of alpha in excercise 1

3.2.2 Exercise 3

Based on your finding in Exercise 2 comment on the relative importance of `alpha` for the performance of the model.

We note that altering alpha slightly from 0 only has a small impact on the rmse of our final model. It is for this reason that we prefer small values of alpha, it may even be the case that we prefer to not include and alpha term in our model and use the standard regression model.

3.2.3 Exercise 4

What happens if you set `alpha = 0`?

If we set `alpha = 0`, then we recover kernelized regression without the penalty term

[15]:
```
alpha = 0.000000000001

m4 = GridSearchCV(
    KernelRidge(kernel='rbf'),
    param_grid={"alpha": np.array([alpha]),
                "gamma": np.logspace(0, 3, num=50)},
    scoring = 'neg_mean_squared_error',
    cv = KFold(5, shuffle=True, random_state=1234)
)

m4 = m4.fit(X,y)

m4.best_params_
```

[15]:
```
{'alpha': 1e-12, 'gamma': 10.985411419875584}
```

---

4 3. Logistic Regression (Introduction)

4.1 3.1 Data

For this introduction we will be using a data set on spam emails from the OpenIntro project, these data are similar in spirit to those used in lecture (which came from Machine Learning: A Probabilistic Perspective). These data have been provided as `email.csv` along with this worksheet. A full data dictionary can be found here. To keep things simple this week we will restrict our exploration to including only the following columns: `spam`, `exclaim_mess`, `format`, `num_char`, `line_breaks`, and `number`. Brief descriptions of these variables are below:

- `spam` - Indicator for whether the email was spam.
- `exclaim_mess` - The number of exclamation points in the email message.

- `format` - Indicates whether the email was written using HTML (e.g. may have included bolding or active links).
- `num_char` - The number of characters in the email, in thousands.
- `line_breaks` - The number of line breaks in the email (does not count text wrapping).
- `number` - Factor variable saying whether there was no number, a small number (under 1 million), or a big number.

[16]:
```
email = pd.read_csv('email.csv')[ ['spam', 'exclaim_mess', 'format',␣
↪'num_char', 'line_breaks', 'number'] ]
email
```

[16]:

	spam	exclaim_mess	format	num_char	line_breaks	number
0	0	0	1	11.370	202	big
1	0	1	1	10.504	202	small
2	0	6	1	7.773	192	small
3	0	48	1	13.256	255	small
4	0	1	0	1.231	29	none
…	…	…	…	…	…	…
3916	1	0	0	0.332	12	small
3917	1	0	0	0.323	15	small
3918	0	5	1	8.656	208	small
3919	0	0	0	10.185	132	small
3920	1	1	0	2.225	65	small

[3921 rows x 6 columns]

Given that our data includes the `number` column which is categorical, we will take care of the necessary dummy coding (specifically one hot encoding here) before we begin to explore and model these data. Here we will use the pandas' `get_dummies` function but a similar outcome can be achieved using sklearn's `OneHotEncoder`.

[17]:
```
email = pd.get_dummies(email)
email
```

[17]:

	spam	exclaim_mess	format	num_char	line_breaks	number_big	\
0	0	0	1	11.370	202	1	
1	0	1	1	10.504	202	0	
2	0	6	1	7.773	192	0	
3	0	48	1	13.256	255	0	
4	0	1	0	1.231	29	0	
…	…	…	…	…	…	…	
3916	1	0	0	0.332	12	0	
3917	1	0	0	0.323	15	0	
3918	0	5	1	8.656	208	0	
3919	0	0	0	10.185	132	0	
3920	1	1	0	2.225	65	0	

	number_none	number_small

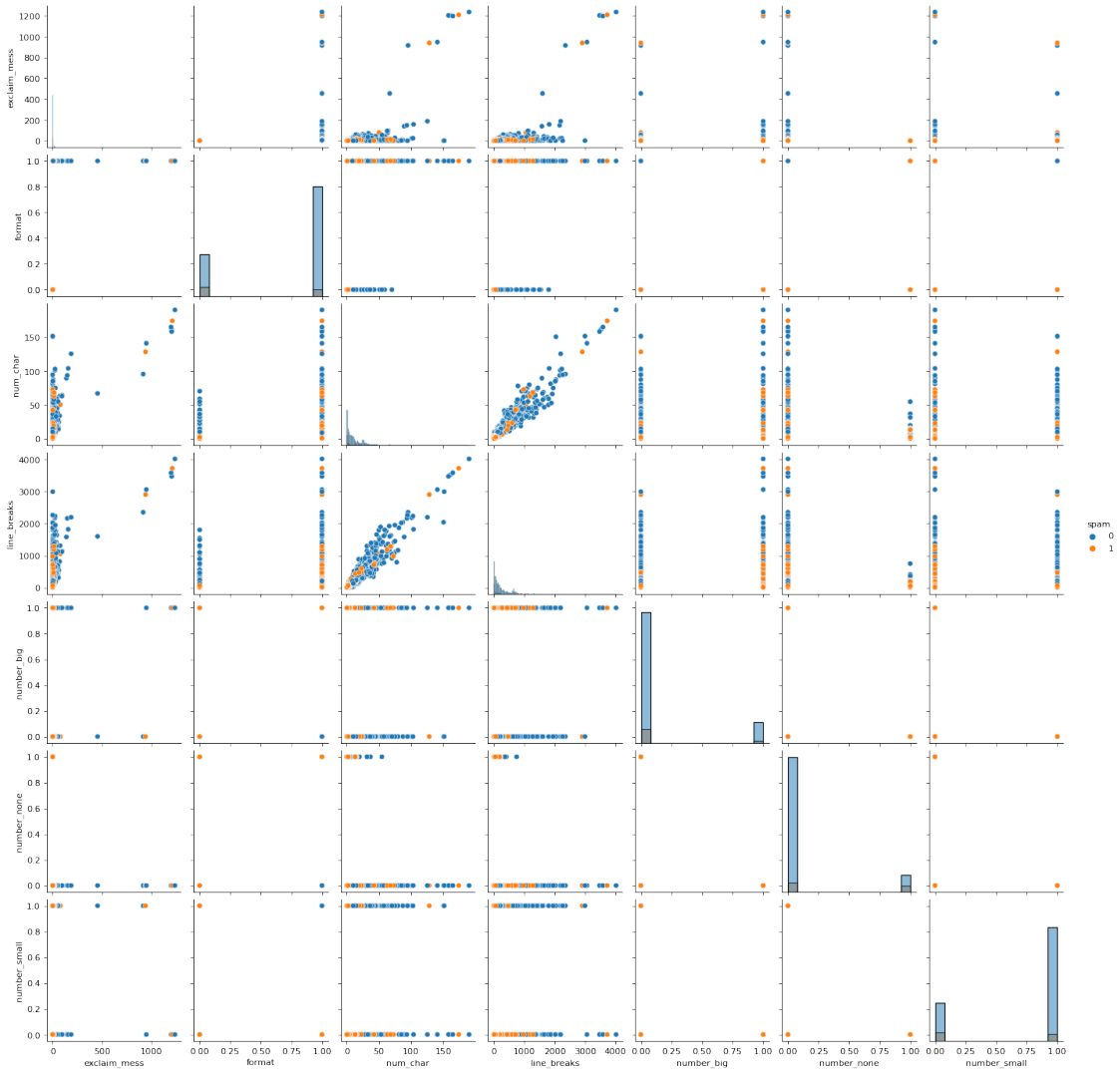
```
0                0              0
1                0              1
2                0              1
3                0              1
4                1              0
…              …              …
3916             0              1
3917             0              1
3918             0              1
3919             0              1
3920             0              1

[3921 rows x 8 columns]
```

4.2 3.2 Exploration

We can construct a pair plot of these data to explore some of the basic relationships between the outcome and the features, since our outcome variable, **spam**, is binary (categorical) we can use it is as the **hue** argument with the **pairplot** function to get a better sense about how spam and non-spam messages differ across our features.

[18]:
```
sns.pairplot(email, hue='spam', diag_kind='hist')
```

[18]:
```
<seaborn.axisgrid.PairGrid at 0x7f4c75a7ba50>
```



4.2.1 Exercise 5

Of the plots provided by the pair plot which are the most useful? Explain.

we can see a strong correlation between the number of characters and line breaks. However, this is not useful in our analysis.

The plots of number_big, number_none and number_small vs num_chars and line breaks give us an idea of which emails are spam. It appears that a low number of line_breaks can indicate a spam email for all of the number sizes.

The bar graphs of number vs number show us that big numbers are not very common and don't indicate a spam or non spam email. There is a relatively large proportion of spam emails when no

number is included in the email. Finally, small numbers are more common in emails, however the number of spam appears similar across both no small number and when a small number is included.

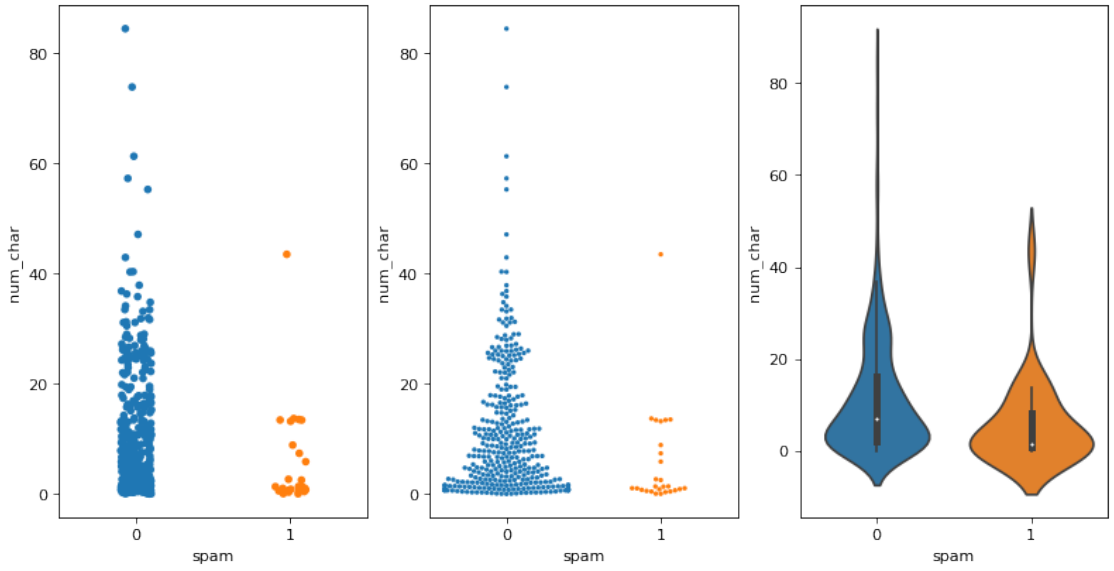
When plotting data with categorical or binary features there are a number of additional plotting methods in seaborn that are useful for exploring potential relationships. In particular, `stripplot`, `swarmplot` and `violinplot` can all provide useful insights while avoiding the overplotting issue.

Another useful type of plot for data like this is known as a strip plot which adds jitter to avoid the overplotting while showing "all" of the data, which is particularly important when your categories are not balanced.

[19]:
```
# Since some methods show all points, we will sample the data to get a more␣
 ↪managable n
# swarmplot in particular is very slow for large n

email_samp = email.sample(frac=0.1, random_state=1234)
```

[20]:
```
plt.figure(figsize=(12,6))

plt.subplot(131)
sns.stripplot(y='num_char', x='spam', data=email_samp)

plt.subplot(132)
sns.swarmplot(y='num_char', x='spam', size=3, data=email_samp)

plt.subplot(133)
sns.violinplot(y='num_char', x='spam', data=email_samp)

plt.show()
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/categorical.py:1296: UserWarning:
8.2% of the points cannot be placed; you may want to decrease the size of the
markers or use stripplot.
  warnings.warn(msg, UserWarning)
```



4.2.2 Exercise 6

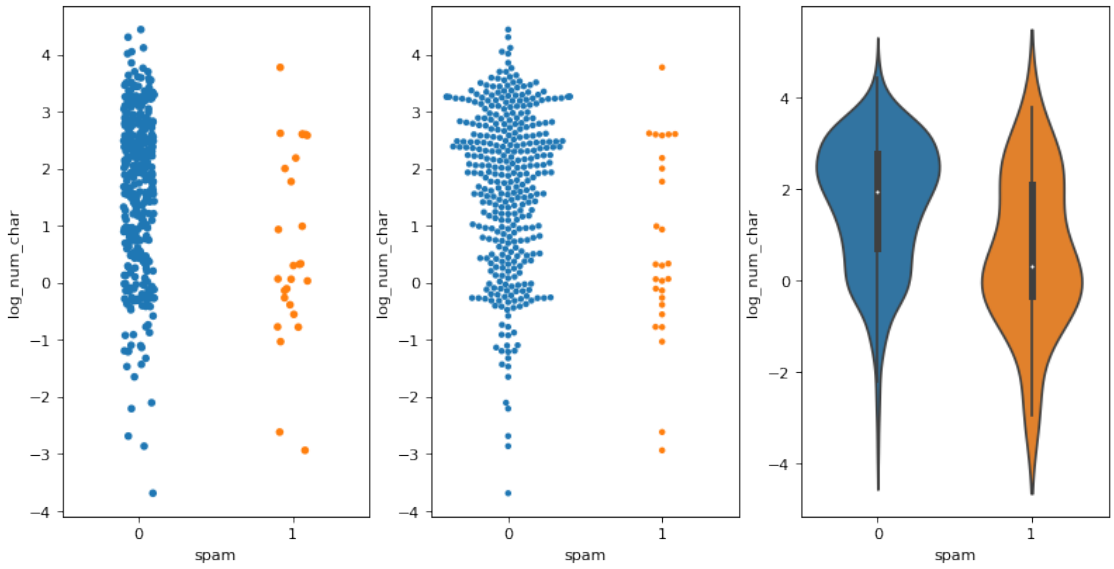
Comment on any structure you see in these plots, does `num_char` appear to have different distributions between the spam and non-spam emails?

It appears that spam emails have a lower mean number of characters than non spam emails and emails including more than 20 characters are rarely spam emails.

Since the `num_char` variable is quite right skewed we can explore using a log transform of the variable before plotting,

[21]:
```
email_samp['log_num_char'] = np.log(email_samp.num_char)
```

[22]:
```
plt.figure(figsize=(12,6))

plt.subplot(131)
sns.stripplot(y='log_num_char', x='spam', data=email_samp)

plt.subplot(132)
sns.swarmplot(y='log_num_char', x='spam', size=4, data=email_samp)

plt.subplot(133)
sns.violinplot(y='log_num_char', x='spam', data=email_samp)

plt.show()
```



4.2.3 Exercise 7

Comment on any structure you see in these plots, does `log_num_char` appear to have different distributions between the spam and non-spam emails?

Now, from the violin plot, we can see that `log_num_char` is distributed more closely around zero for spam emails than non-spam emails. A much larger proportion of spam emails have negative values under the log transformation than for non-spam emails.

4.3 3.2 Model Fitting

Logistic (and multinomial) regression models are fit using the `LogisticRegression` model function from the `linear_model` submodule of sklearn.

[23]:
```
from sklearn.linear_model import LogisticRegression
```

[24]:
```
y = email.spam
X = email.drop('spam', axis=1)
```

There are a couple of "unique" properties of this model implementation in sklearn that are important to be aware of. The first is that the model can be fit with regularization (i.e. an ℓ_2 or ℓ_1 penalty) but the inclusion of the ℓ_2 penalty is the default behavior for these models.

For those coming from other programming / modeling languages this may be somewhat surprising, this strange default is probably the most common reason that results from sklearn might not

immediately match results from other tools. This behavior can be explicitly controlled via the `penalty` argument. Note that if you did wish to include a penalty on the coefficients then just like with ridge or lasso it is necessary to tune this penalty parameter. However, `LogisticRegression` does not use `alpha` for this tuning parameter but instead uses `C` which is the inverse of the `alpha` we have been using - i.e. smaller values of `C` imply more regularization.

Otherwise, this model function behaves as all of the other model functions we have seen so far, below we fit the model and then extract the fitted coefficients.

[25]:
```
# Note we use `fit_intercept = False` to avoid a rank deficient model matrix␣
↪sicne
# we used one-hot encoding for the `number` feature.

m = LogisticRegression(penalty = 'none', fit_intercept = False, solver='lbfgs',␣
↪max_iter=1000).fit(X, y)
```

[26]:
```
# This is just some fancy python code for attaching the column names to
# their corresponding coefficeint values

dict(zip(X.columns, zip(*m.coef_)))
```

[26]:
```
{'exclaim_mess': (0.009566215089042278,),
 'format': (-0.6069073689343712,),
 'num_char': (0.05454171666804308,),
 'line_breaks': (-0.0054627389638335746,),
 'number_big': (-1.2634907762440883,),
 'number_none': (-0.7079566684721311,),
 'number_small': (-1.9510206230957052,)}
```

4.3.1 Exercise 8

Based on these results, interpret the coefficient for `num_char` in context.

Firstly, we note that `num_char` has a very small coefficient in terms of absolute value relative to the other variables (excluding `exclaim_mess` and `line_breaks`). This would lead us to believe that `num_chars` has little predictive power in aiding our regression model predicting spam emails.

This matches our intuition from exercises 6 and 7, as there it appeared that the distributions differed slightly. However, there isn't a great difference between the distributions of `num_chars` for spam and non-spam emails. Indicating that it may not be overly important in predicting spam emails where other more predictive variables are available.

4.3.2 Exercise 9

Generally comment on how each feature is related to the probability of an email being spam.

Since a prediction of 0 represents a non-spam email and 1 a spam email, the coefficient represents the change in the probability of an email being spam for a change of 1 unit in the corresponding variable. Negative coefficients also indicate a negative effect on the probability of an email being spam.

Larger coefficinets in our regression model generally indicate a greater contribution of that feature to an email being a spam email. Note that this general rule applies when the variables are of a similar scale, in our case `num_chars`, `line_breaks` and `exclaim_mess` all contain very large values relative to the other variables and so will have smaller coefficients.

4.4 3.3 Model Predictions

Unlike with our previous regression models, the fitted `LogisticRegression` objects provide multiple prediction methods. Specifically: `predict` which predicts the class label (either `0` or `1`), `predict_proba` which predicts the class probabilities, and `predict_log_proba` which predicts the log probabilities of each class.

[27]:
```
m.predict(X.head())
```

[27]:
```
array([0, 0, 0, 0, 0])
```

[28]:
```
m.predict_proba(X.head())
```

[28]:
```
array([[0.91323042, 0.08676958],
       [0.95601516, 0.04398484],
       [0.95792936, 0.04207064],
       [0.9409632 , 0.0590368 ],
       [0.68776026, 0.31223974]])
```

[29]:
```
m.predict_log_proba(X.head())
```

[29]:
```
array([[-0.09076706, -2.44449915],
       [-0.0449815 , -3.12391036],
       [-0.04298125, -3.16840506],
       [-0.06085124, -2.82959435],
       [-0.37431496, -1.16398399]])
```

Here the class label prediction is based on which of the classes has the largest probability. In this case the model is predicting that all five of these messages are not spam (`spam==0`).

4.4.1 Exercise 10

According to this model, what is the probability that a message with 5 exclamation marks, unformated, with 5000 characters, 10 line breaks, and no numbers was spam?

[30]:
```
#m.predict([5,0,5000,10,0,1,0])
# ['spam', 'exclaim_mess', 'format', 'num_char', 'line_breaks', 'number']
data2 = { 'exclaim_mess' : 5, 'format' : 0 , 'num_char' : 5000, 'line_breaks' :␣
 ↪10 , 'number_big' : 0, 'number_none' : 1, 'number_small' : 0}
data = pd.DataFrame([data2])

print(m.predict(data))
print(m.predict_proba(data))
```

```
[1]
[[0. 1.]]
```

The Model is predicting Spam and it gives a probability of 1.

4.5 3.3 Model Validation

4.5.1 3.3.1 Confusion Matrix

As our outcome variable of interest is binary, either 0 or 1, using metrics like root mean squared error does not make sense. Instead we use metrics that measure our classifier's ability to correctly label our data using the following possible outcomes for each prediction:

- *true positives* - labeled as spam when it is spam
- *false positives* - labeled as spam when it is not spam
- *true negatives* - labeled as not spam when it is not spam
- *false negatives* - labeled as not spam when it is spam

The `predict` function chooses these labels based on whichever probability is larger, in the binary case this is equivalent to asking if the probability of being spam is ≥ 0.5 for each observation. This choice of threshold however is arbitrary and we can use any value between 0 and 1 for determining what we will label spam vs not spam. Below we engage is some data book keeping and then define the `confusion_plot` function for visualizing these different outcomes for the different possible threshold values.

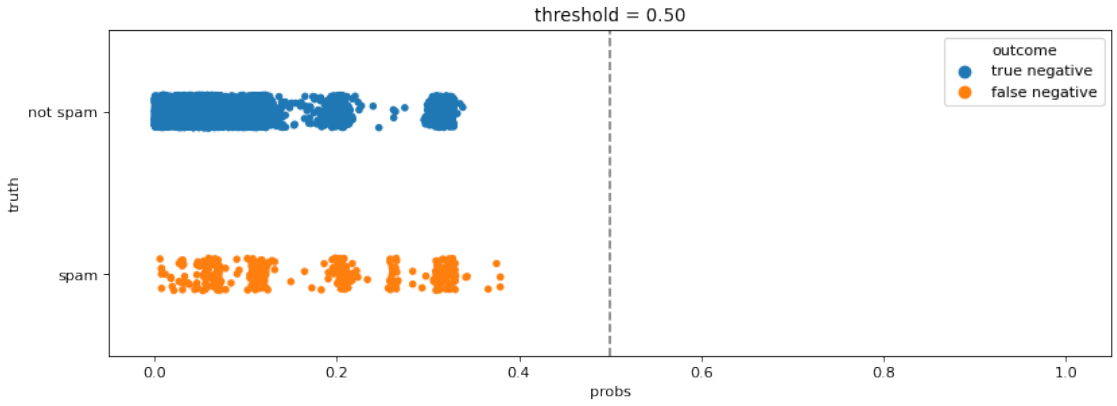
[31]:
```
# This transformation is necessary so that seaborn behaves correctly when␣
 ↪plotting the data horizontally
truth = pd.Categorical.from_codes(y, categories = ('not spam','spam'))
probs = m.predict_proba(X)[:,1]
```

[32]:
```
def confusion_plot(threshold=0.5):
    d = pd.DataFrame(
        data = {'spam': y, 'truth': truth, 'probs': probs}
    )

    # Create a column called outcome that contains the labeling outcome
    # for the given threshold
    d['outcome'] = 'other'
```

```
    d.loc[(d.spam == 1) & (d.probs >= threshold), 'outcome'] = 'true positive'
    d.loc[(d.spam == 0) & (d.probs >= threshold), 'outcome'] = 'false positive'
    d.loc[(d.spam == 1) & (d.probs <  threshold), 'outcome'] = 'false negative'
    d.loc[(d.spam == 0) & (d.probs <  threshold), 'outcome'] = 'true negative'

    # Create plot and color according to outcome
    plt.figure(figsize=(12,4))
    plt.xlim((-0.05,1.05))
    sns.stripplot(y='truth', x='probs', hue='outcome', data=d)
    plt.axvline(x=threshold, linestyle='dashed', color='black', alpha=0.5)
    plt.title("threshold = %.2f" % threshold)
    plt.show()

    return sklearn.metrics.confusion_matrix(y_true=d.spam, y_pred=d.probs >=␣
↪threshold)
```

Below we run this function using a threshold of 0.5 to examine how the logistic regression classifier is performing. Feel free to vary this between 0 and 1 and observe the effect.

[33]:
```
confusion_plot(threshold=0.5)
```



[33]:
```
array([[3554,    0],
       [ 367,    0]])
```

The small matrix that is returned by the function is the confusion matrix for the given threshold and it contains the number of each outcome (in the same order as presented in the graph), this is calculated use the `confusion_matrix` function from the `sklearn.metrics` submodule.

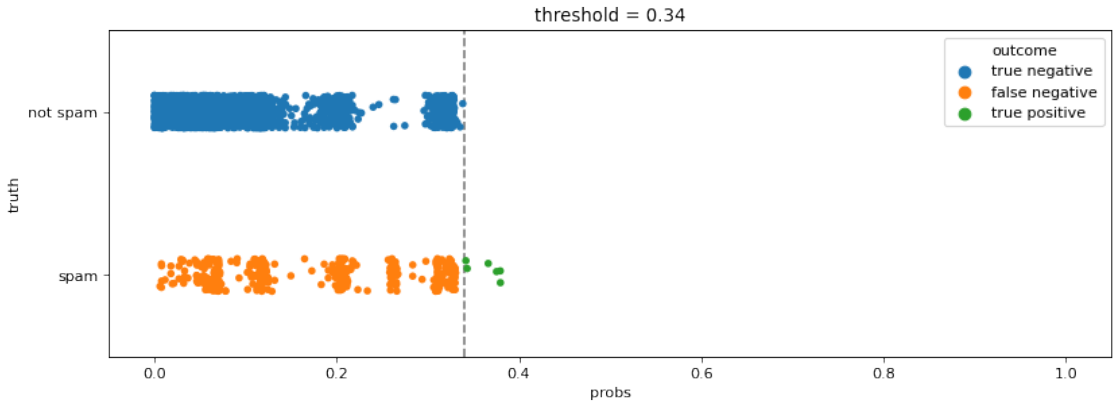
4.5.2 Exercise 11

Based on these results what value of threshold for this model might be a reasonable choice? Think about the relative cost of a false negative vs a false positive.

The relative cost of a false positive is higher than for a false negative. We would prefer to label some emails as spam and not have to check our spam folder, than have important emails labeled as spam and we never see them.

Therefore, we may prefer a value of

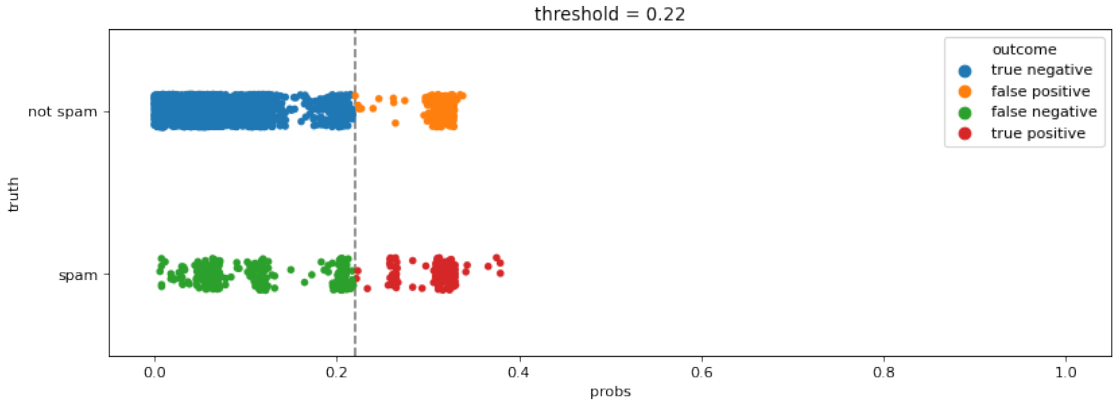
[34]:
```
confusion_plot(threshold=0.34)
```



[34]:
```
array([[3554,    0],
       [ 361,    6]])
```

In this case, we do not miss any important emails. However, the model is rather ineffective on this data set, predicting the vast majority of emails as non-spam. Therefore, we may prefer a model that sacrifices some of our non-spam emails and predicts more spam emails. A value of `threshold = 0.22` achieves this, however it leads to a large number of false positives which are expensive for our user.

[35]:
```
confusion_plot(threshold=0.22)
```



[35]:
```
array([[3310,  244],
       [ 254,  113]])
```

If I was the user, I would prefer a different model with improved predictive accuracy and neither of these values would be suitable.

4.5.3 3.3.2 ROC curves

Another method for seeing the effect of these different choices of threshold is to generate a Receiver operating characteristic (ROC) curve for the model which plots the true positive rate against the false positive rate for different threshold values.

[36]:
```
from sklearn.metrics import roc_curve, precision_recall_curve
```

This function returns a tuple of the false positive rate, true positive rate, and threshold values based on the true labels and the predicted probabilities.

Here we take those results and construct a data frame to enable us to create the ROC plot using seaborn.

[37]:
```
roc_calc = roc_curve(y_true=y, y_score=probs)

roc = pd.DataFrame(
    data = np.c_[roc_calc],
    columns = ('false positive rate', 'true positive rate', 'threshold')
)

roc
```

[37]:

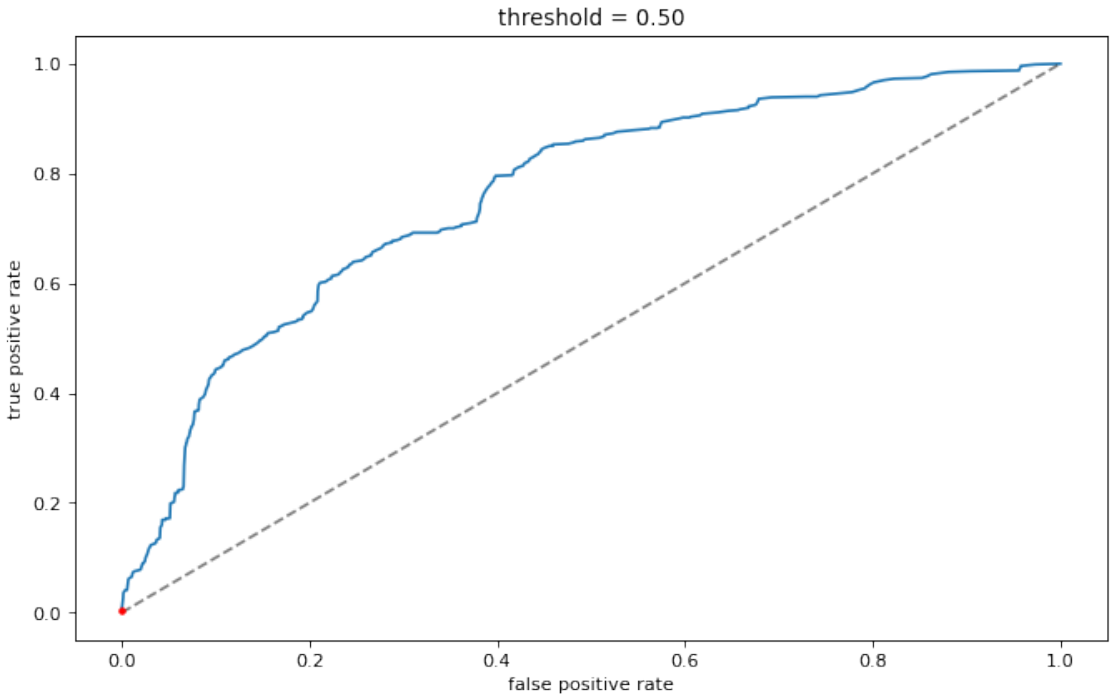
	false positive rate	true positive rate	threshold
0	0.000000	0.000000	1.379605
1	0.000000	0.002725	0.379605

```
2              0.000000            0.016349   0.341750
3              0.000844            0.016349   0.332233
4              0.000844            0.019074   0.330519
..                  …                   …          …
586            0.957794            0.994550   0.007774
587            0.957794            0.997275   0.007761
588            0.973270            0.997275   0.006294
589            0.973270            1.000000   0.006194
590            1.000000            1.000000   0.000023

[591 rows x 3 columns]
```

Using this data frame, the function below creates the ROC curve (blue) via a seaborn lineplot, while the remaining code draws the point on the ROC curve that corresponds to the given threshold value (red), and finally the 0-1 line (grey, dashed). The 0-1 line is the ROC curve expected for a model that determines labels by chance (e.g. flipping a coin with `heads` is spam, `tails` is not spam as a model).

[38]:
```
def roc_plot(threshold=0.5):
    i = (np.abs(roc.threshold - threshold)).idxmin()

    sns.lineplot(x='false positive rate', y='true positive rate', data=roc,␣
 ↪ci=None)

    plt.plot([0,1],[0,1], 'k--', alpha=0.5) # 0-1 line
    plt.plot(roc.iloc[i,0], roc.iloc[i,1], 'r.')

    plt.title("threshold = %.2f" % threshold)
    plt.show()
```

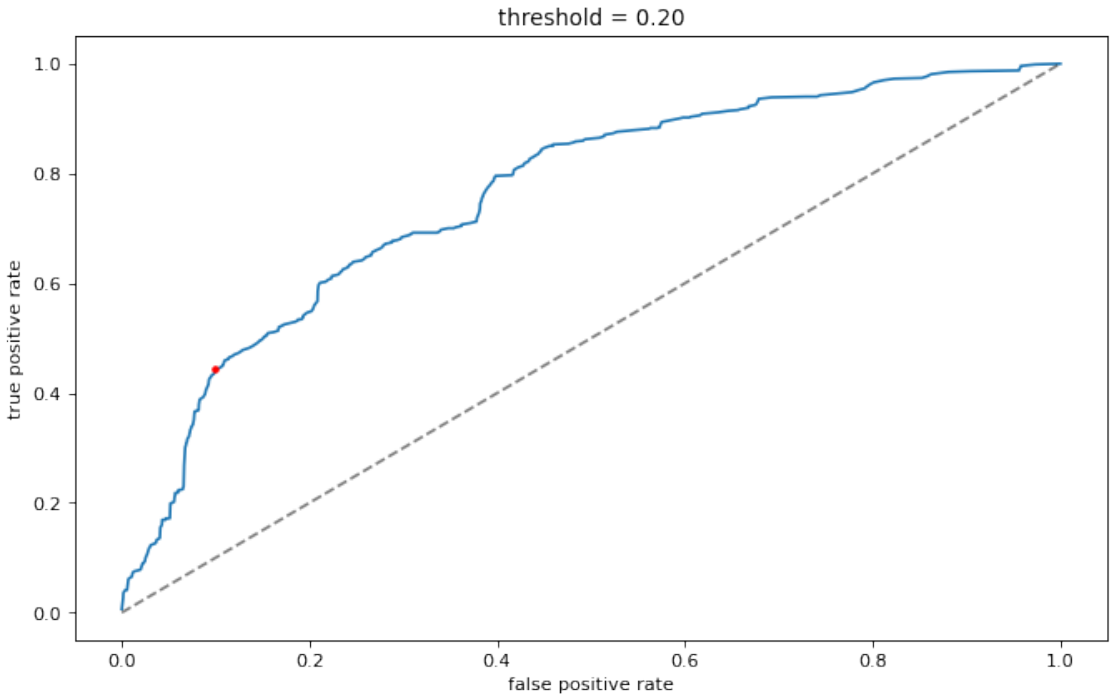
[39]:
```
roc_plot(threshold=0.5)
```



4.5.4 Exercise 12

Try adjusting the threshold value in `roc_plot`, what happens to the red point as this value is changed?

[40]:
```
roc_plot(threshold=0.2)
```



We note that with `threshold = 0.5`, we have

$$\text{recall} = \text{precision} = 0$$

as the model predicts all emails to be non-spam. As we decrease the value of the threshold, we classify more emails as spam and eventually we have

$$\text{recall} \to 1 \text{ and precision} \to 1.$$

Therefore, the red point moves along the curve to the top right as we reduce the threshold.

4.5.5 3.3.3 Area under the curve (AUC)

Finally, a common summary statistic that is used to summarize the performance of a binary classifier over all the possible thresholds is the area under the curve (AUC), which as the name implies, is the area under the ROC curve. This can be calculated from the prediction results using the `roc_auc_score` function from the `sklearn.metrics` submodule.

[41]:
```
sklearn.metrics.roc_auc_score(y_true=y, y_score=probs)
```

[41]:
```
0.7607075115117632
```

Beyond giving a simple numeric summary statistic for a classifier, the AUC also has a useful alternative interpretation - the AUC also equals the probability that the classifier will rank a

randomly chosen positive case (`spam` here) higher than a randomly chosen negative case (`not spam` here).

4.5.6 Exercise 13

Based on these and the preceeding results, how well does this classifier perform? Justify your answer.

The classifier is ranking a randomly chosen spam email higher than a randomly chosen non spam email with probability 0.76. This appears to be a high probability, however, the classifier doesn't provide great results for any chosen value of the threshold - as discussed in exercise 11.

I would conclude that this classifier would need to be improved before being implemented in any email system.

Note that the AUC does not tell us what value of threshold should be used for our classifier, just its overall performance. If we would like to choose an optimal threshold value for practical decision making then we need to define a specific loss function for our data / problem. Specifically, we need to have some numeric measure of the benefit for true positives and true negatives and the costs of false negatives and false positives. For example the cost of a false positive vs. false negative is very different for this spam example than it would be for a medical diagnostic test like a Cancer screening test.

4.6 4. Competing the worksheet

At this point you have hopefully been able to complete all the preceeding exercises. Now is a good time to check the reproducibility of this document by restarting the notebook's kernel and rerunning all cells in order.

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF and turn it in on gradescope under the `mlp-week06` assignment.

[42]:
```
!jupyter nbconvert --to pdf mlp-week06.ipynb
```

```
[NbConvertApp] Converting notebook mlp-week06.ipynb to pdf
[NbConvertApp] Support files will be in mlp-week06_files/
[NbConvertApp] Making directory ./mlp-week06_files
[NbConvertApp] Making directory ./mlp-week06_files
[NbConvertApp] Making directory ./mlp-week06_files
[NbConvertApp] Writing 84746 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', './notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', './notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
```

```
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 190784 bytes to mlp-week06.pdf
```

Created in Deepnote