دانشگاه صنعتی خواجه نصیرالدین طوسی
دانشکدهٔ مهندسی برق - گروه مهندسی کنترل

درس یادگیری ماشین

گزارش مینی پروژه ۳

نام و نام خانوادگی	دانیال عبداللهی نژاد
شمارهٔ دانشجویی	۴۰۱۰۹۱۶۴
تاریخ	خردادماه ۱۴۰۳

فهرست مطالب

۱ سوال اول ۳
۱.۱ . . . . . ۳
۲.۱ . . . . . ۱۰
۳.۱ . . . . . ۱۰
۴.۱ . . . . . ۱۸

۲ سوال سوم ۲۶
۱.۲ . . . . . ۲۶
۲.۲ . . . . . ۲۷
۳.۲ . . . . . ۲۹
۴.۲ . . . . . ۳۲
۵.۲ . . . . . ۳۵
۶.۲ . . . . . ۳۶

فهرست تصاویر

۱ نام و تعداد ویژگی‌ها و ابعاد دیتاست Iris . . . ۳
۲ میانگین و واریانس ویژگی‌ها . . . ۴
۳ نمایش توزیع ویژگی‌ها . . . ۵
۴ نمودار جعبه‌ای داده‌ها . . . ۵
۵ ماتریس همبستگی دیتاست Iris . . . ۶
۶ بصری‌سازی به کمک t-SNE . . . ۸
۷ بصری‌سازی به کمک PCA . . . ۹
۸ واریانس توضیح داده شده با توجه به Principal Components . . . ۹
۹ نتایج طبقه‌بند SVM با کرنل خطی . . . ۱۰
۱۰ ماتریس درهم ریختگی طبقه‌بند SVM با کرنل خطی . . . ۱۱
۱۱ ناحیه تصمیم‌گیری طبقه‌بند SVM با کرنل خطی با کاهش بعد . . . ۱۱
۱۲ نمودار دقت با توجه به درجه کرنل . . . ۱۳
۱۳ ماتریس درهم ریختگی با توجه به درجه کرنل . . . ۱۴
۱۴ ناحیه تصمیم‌گیری طبقه‌بند با توجه به درجات مختلف poly . . . ۱۶
۱۵ polynomial ۳ درجه اول به صورت دستی . . . ۲۳
۱۶ polynomial ۳ درجه دوم به صورت دستی . . . ۲۴
۱۷ polynomial ۴ درجه سوم به صورت دستی . . . ۲۵
۱۸ مودار دقت با توجه به درجه به صورت دستی . . . ۲۶
۱۹ فلوچارت مدل طبقه‌بندی . . . ۲۷
۲۰ معماری شبکه Autoencoder . . . ۲۸
۲۱ معماری شبکه عصبی برای طبقه‌بندی . . . ۲۹
۲۲ نمودار پراکندگی برچسب‌ها . . . ۳۰
۲۳ پراکندگی برچسب‌ها . . . ۳۰
۲۴ ماتریس درهم‌ریختگی . . . ۳۳
۲۵ گزارش طبقه‌بندی با معیارهای مختلف . . . ۳۳
۲۶ نمودار accuracy در برابر recall . . . ۳۵
۲۷ نمودار accuracy در برابر recall برای آستانه‌های متفاوت oversampling . . . ۳۶
۲۸ نتایج طبقه‌بندی در حالت دوم . . . ۳۶
۲۹ ماتریس درهم‌ریختگی در حالت دوم . . . ۳۷

۱ سوال اول

۱.۱

مجموعه‌داده Iris یکی از مجموعه‌داده‌های ساده و پرکاربرد در زمینه آموزش ماشین یادگیری است. این مجموعه‌داده شامل ۱۵۰ نمونه می‌باشد که هر نمونه دارای چهار ویژگی عددی است: طول و عرض گلبرگ و طول و عرض کاسبرگ. این ویژگی‌ها برای طبقه‌بندی داده‌ها به سه گونه مختلف از گیاه Iris به‌کار می‌روند: Iris-setosa، Iris-versicolor و Iris-virginica.

هر گونه از گیاه Iris دارای ۵۰ نمونه است که به طور مساوی در مجموعه‌داده توزیع شده‌اند. ویژگی‌های عددی ذکر شده به صورت پیوسته اندازه‌گیری شده و در طبقه‌بندی این سه گونه نقش کلیدی دارند. هدف اصلی استفاده از این مجموعه‌داده، توسعه و ارزیابی مدل‌های یادگیری ماشین برای مسائل طبقه‌بندی است.

داده را به صورت زیر از طریق کتابخانه sklearn فراخوانی می‌کنیم:

```
iris = datasets.load_iris()

df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['species'] = iris.target
df['species_name'] = df['species'].map(dict(enumerate(iris.target_names)))
```

همان‌طور که مشاهده می‌شود، کلاس داده‌ها هم به صورت عددی و هم به صورت categorical نمایش داده شده‌اند. علاوه بر این، ۴ ویژگی برای داده‌ها داریم. در کل، داده‌ها شامل ۱۵۰ نمونه می‌باشند که ابعاد داده و نام ویژگی‌ها را می‌توان در شکل ۱ مشاهده کرد. همچنین balanced بودن دیتاست را بررسی می‌کنیم، که مشاهده می‌شود نمونه‌ها کاملاً balanced هستند و نیازی به استفاده از روش‌هایی مانند Undersampling نمی‌باشد.

```
print(iris.feature_names)
print(len(iris.feature_names))
print(df.shape)
df['species_name'].value_counts()
```

✓ 0.0s

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
4
(150, 6)

species_name
setosa        50
versicolor    50
virginica     50
Name: count, dtype: int64
```

شکل ۱: نام و تعداد ویژگی‌ها و ابعاد دیتاست Iris

در مرحله بعد، به محاسبه میانگین و واریانس ویژگی‌ها می‌پردازیم که در شکل ۲ نشان داده شده است.

همان‌طور که در شکل ۲ مشخص است، میانگین طول کاسبرگ حدود ۶ سانتی‌متر، میانگین عرض کاسبرگ ۳ سانتی‌متر، میانگین طول گلبرگ حدود ۴ سانتی‌متر و میانگین عرض گلبرگ ۱ سانتی‌متر می‌باشد. از نظر واریانس، ویژگی طول گلبرگ، طول کاسبرگ، عرض گلبرگ و عرض کاسبرگ به ترتیب بیشترین تا کمترین انحراف معیار را دارند. علاوه بر این، بیشینه و کمینه این مقادیر نیز در شکل ۲ آمده است.

```
df.describe()
```
✓ 0.0s

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

```
mean = df.groupby('species_name').mean()
mean
```
✓ 0.0s

species_name	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
setosa	5.006	3.428	1.462	0.246	0.0
versicolor	5.936	2.770	4.260	1.326	1.0
virginica	6.588	2.974	5.552	2.026	2.0

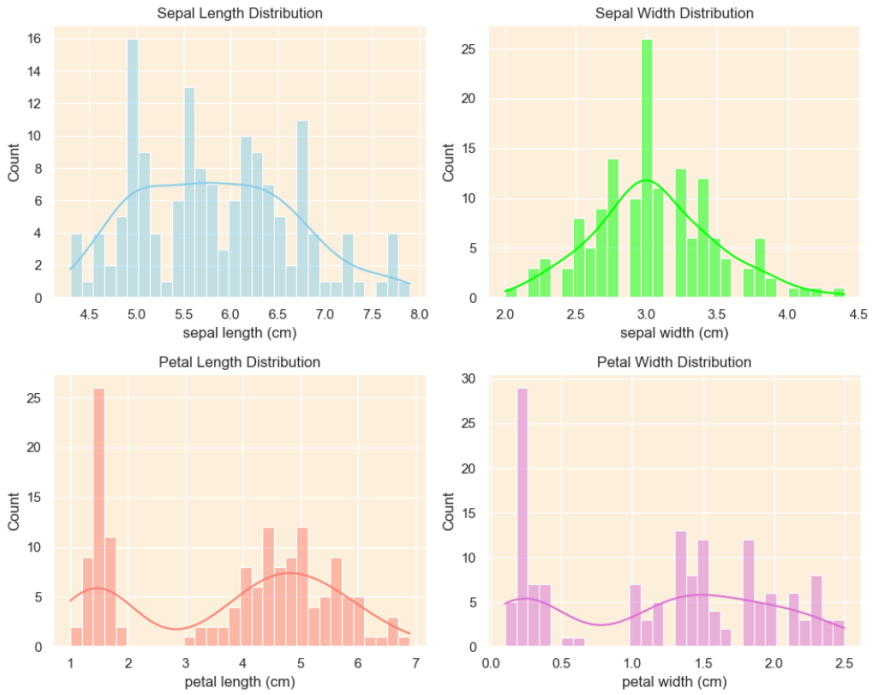
```
med = df.groupby('species_name').median()
med
```
✓ 0.0s

species_name	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
setosa	5.0	3.4	1.50	0.2	0.0
versicolor	5.9	2.8	4.35	1.3	1.0
virginica	6.5	3.0	5.55	2.0	2.0

شکل ۲: میانگین و واریانس ویژگی‌ها

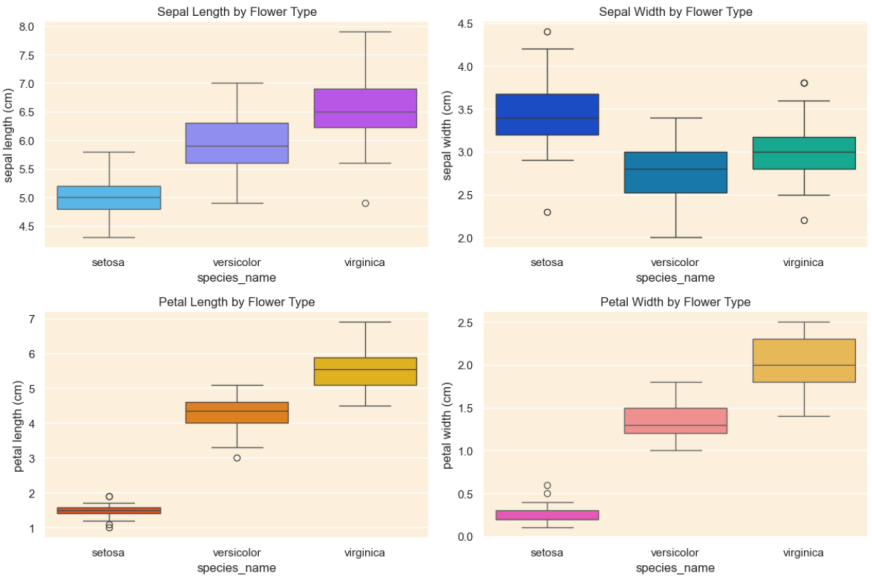
در شکل ۲ همچنین میانگین و میانه این ویژگی‌ها به تفکیک هر کلاس آمده است. از این شکل می‌توان نتیجه گرفت که کلاس Iris-versicolor بیشترین طول کاسبرگ را دارد و در رتبه دوم Iris-virginica قرار دارد. نکته مهم دیگر این است که در ویژگی عرض گلبرگ، سه کلاس از نظر میانگین اختلاف قابل توجهی دارند و می‌توان پیش‌بینی کرد که با استفاده از این ویژگی، کلاس Iris-setosa را به خوبی تشخیص داد.

در ادامه و به کمک نمودار میله‌ای، توزیع ویژگی‌ها در داده را به نمایش می‌گذاریم. این توزیع در شکل ۳ آمده است. در شکل ۳ مشخص است که طول و عرض کاسبرگ‌ها توزیع تقریباً نرمال با یک قله یا peak دارند در صورتی که توزیع طول و عرض گلبرگ‌ها یک توزیع Bimodel دارند به صورتی که دو قله در آن‌ها وجود دارد و با توجه به این موضوع احتمالاً این دو ویژگی از اهمیت بیشتری برخوردار باشند و می‌توانند حداقل دو کلاس از داده‌های ما را به خوبی جدا کنند.



شکل ۳: نمایش توزیع ویژگی‌ها

در مرحله بعد با دسته‌بندی داده‌ها با توجه به هر کلاس، نمودار جعبه‌ای آن‌ها را رسم می‌کنیم که نگاهی دیگر به توزیع داده‌ها است. نتایج را در شکل ۴ مشاهده می‌کنیم.

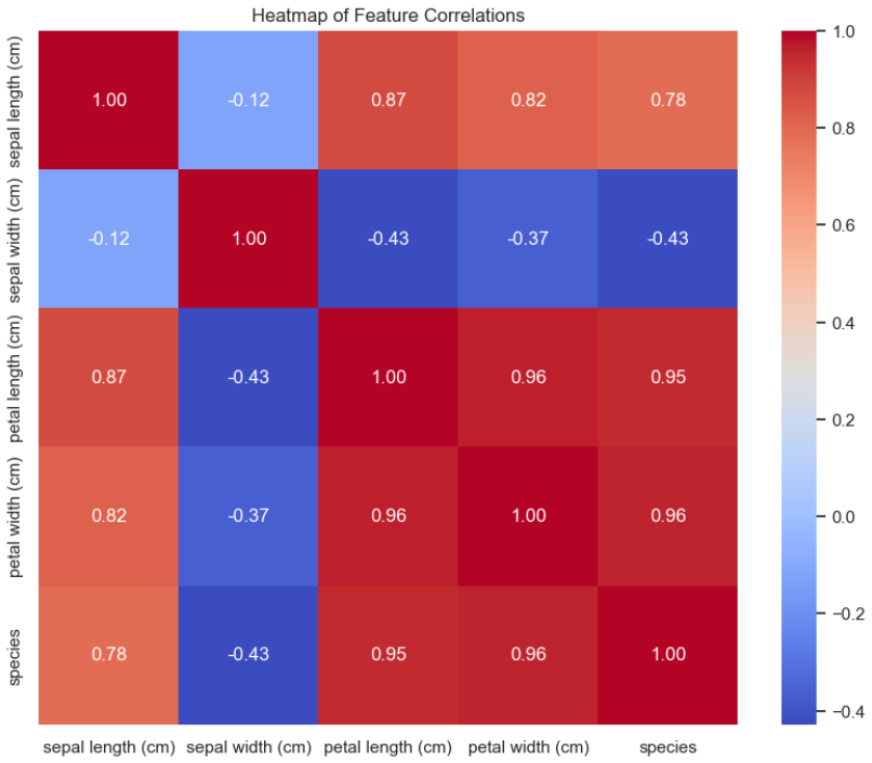


شکل ۴: نمودار جعبه‌ای داده‌ها

مطابق شکل ۴ نکته‌ای که در نمودار میله‌ای بود نیز دیده می‌شود. یعنی اینکه با کمک ویژگی مربوط به گلبرگ، می‌توان به خوبی کلاس‌ها را از هم جدا کرد و این ویژگی‌ها می‌توانند ابزار قدرتمندی برای استفاده در مدل باشند. همان‌طور که دیده می‌شود کلاس Iris-

setosa به خوبی از بقیه کلاس‌ها قابل جدا شدن است و انتظار می‌رود که با دقت ۱۰۰ درصد بتوانیم این کلاس را از بقیه کلاس‌ها جدا کنیم. دو کلاس دیگر از نظر ویژگی‌های مربوط به کاسبرگ نزدیکی زیادی به هم دارند ولی در ویژگی مربوط به گلبرگ می‌توان دید که این دو کلاس را می‌توان با دقت قابل قبولی از هم جدا کرد.

در مرحله بعد در شکل ۵ ماتریس همبستگی مربوط به دیتاست آمده است.



شکل ۵: ماتریس همبستگی دیتاست Iris

با توجه به شکل ۵ دیده می‌شود که بهترین ویژگی‌هایی که با لیبل ما بیشترین همبستگی را دارند به ترتیب عرض گلبرگ، طول گلبرگ، طول کاسبرگ و عرض کاسبرگ هستند. علاوه بر این دیده می‌شود که دو ویژگی عرض گلبرگ و طول گلبرگ به ضریب ۹۶.۰ با یکدیگر هم‌بستگی خطی دارند.

در مرحله بعد از t-SNE استفاده می‌کنیم تا داده را بصری‌سازی کنیم. به طور کلی t-SNE یک روش غیرخطی برای به نمایش گذاشتن داده‌ها با ابعاد بالا است که از روش شباهت (similarity) بین داده‌ها استفاده می‌کند و با ادغام نظریه‌های احتمالاتی سعی می‌کند یک بصری‌سازی مناسب از داده انجام دهد. با انجام این الگوریتم می‌توانیم کلاسترها و الگوهایی که ممکن است در ابعاد بالاتر پنهان شوند را ببینیم. در واقع با بصری‌سازی داده‌ها می‌توان outlier و anomaly مربوط به داده را بهتر مشاهده کرد. نکته منفی این روش این است که در ابعاد بالا می‌تواند از نظر توان محاسباتی بسیار ما را درگیر کند و در این حالات بهتر است از روش‌های خطی مانند PCA استفاده کنیم. در PCA الگوریتم سعی می‌کند جهت یا همان Principle Component هایی را پیدا کند که در آن واریانس داده‌ها ماکسیمایز می‌شود و این Component ها نسبت به هم عمود و uncorrelated هستند. در این الگوریتم ماتریس کوواریانس داده‌ها حساب می‌شود و سپس به کمک بردارهای ویژه و مقادیر ویژه، داده به k بردار ویژه این فضا نگاشت می‌شود.

الگوریتم t-SNE دو هایپرپارامتر مهم دارد که پایین آورده شده است.

```
from sklearn.manifold import TSNE
```

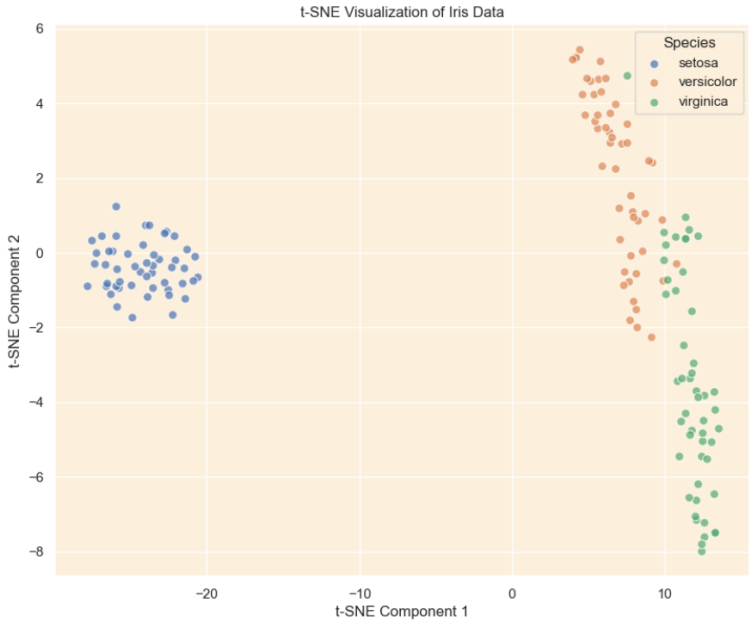
```

df_tsne = df.copy()

tsne = TSNE(n_components=2, perplexity=30, random_state=64)
X_tsne = tsne.fit_transform(df[iris.feature_names])

df_tsne['tsne-2d-one'] = X_tsne[:, 0]
df_tsne['tsne-2d-two'] = X_tsne[:, 1]

plt.figure(figsize=(10, 8))
colors = ['C' + str(i) for i in range(len(iris.target_names))]
for target_name, color in zip(iris.target_names, colors):
    indices_to_plot = df_tsne['species_name'] == target_name
    plt.scatter(df_tsne.loc[indices_to_plot, 'tsne-2d-one'],
                df_tsne.loc[indices_to_plot, 'tsne-2d-two'],
                label=target_name, s=50, alpha=0.7, edgecolors='w')

plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.title('t-SNE Visualization of Iris Data')
plt.legend(title='Species')
plt.grid(True)
plt.show()
```

اولین هایپرپارامتر n-components است که تعداد بعدی است که می‌خواهیم دیتا را به آن نگاشت کنیم. هایپرپارامتر دوم perplexity است که تعداد نزدیکترین همسایگان که در سایر الگوریتم‌های یادگیری چندگانه استفاده می‌شود.

در شکل ۶ این بصری‌سازی به کمک t-SNE آورده شده است.

علاوه بر این، از PCA نیز به صورت زیر استفاده شده تا داده بصری‌سازی شود. توجه شود که این الگوریتم به صورت دستی پیاده شده مطابق روشی که بالاتر توضیح داده شد. ابتدا ماتریس کوواریانس داده‌های استاندارد شده حساب شده و سپس مقادیر و بردارهای ویژه بدست آورده شده سپس از بزرگترین به کمترین مرتب شده و ۲ بردار ویژه بزرگ (دو بعد) انتخاب شده و به کمک این دو بردار ویژه دیتا مپ شده است.

```
X = df[iris.feature_names].values
X_mean = np.mean(X, axis=0)
X_std = np.std(X, axis=0)
X_standardized = (X - X_mean) / X_std

cov_matrix = np.cov(X_standardized, rowvar=False)

eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

sorted_index = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[sorted_index]
sorted_eigenvectors = eigenvectors[:, sorted_index]

```

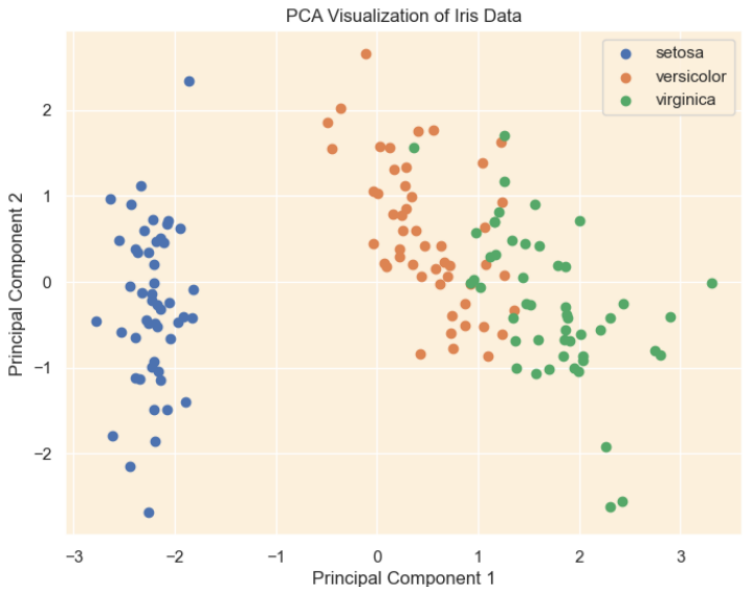


شکل ۶: بصری‌سازی به کمک t-SNE

```python
# Select the top k eigenvectors (here we select top 2 for 2D visualization)
k = 2
eigenvector_subset = sorted_eigenvectors[:, 0:k]
X_reduced = np.dot(X_standardized, eigenvector_subset)

df_pca = pd.DataFrame(data=X_reduced, columns=['PC1', 'PC2'])
df_pca['species'] = df['species']
df_pca['species_name'] = df['species'].map(dict(enumerate(iris.target_names)))

plt.figure(figsize=(8, 6))
colors = ['C' + str(i) for i in range(len(iris.target_names))]
for target_name, color in zip(iris.target_names, colors):
    indices_to_plot = df_pca['species_name'] == target_name
    plt.scatter(df_pca.loc[indices_to_plot, 'PC1'],
                df_pca.loc[indices_to_plot, 'PC2'],
                label=target_name)

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA Visualization of Iris Data')
plt.legend()
plt.show()
```

نتایج در شکل ۷ آمده است.

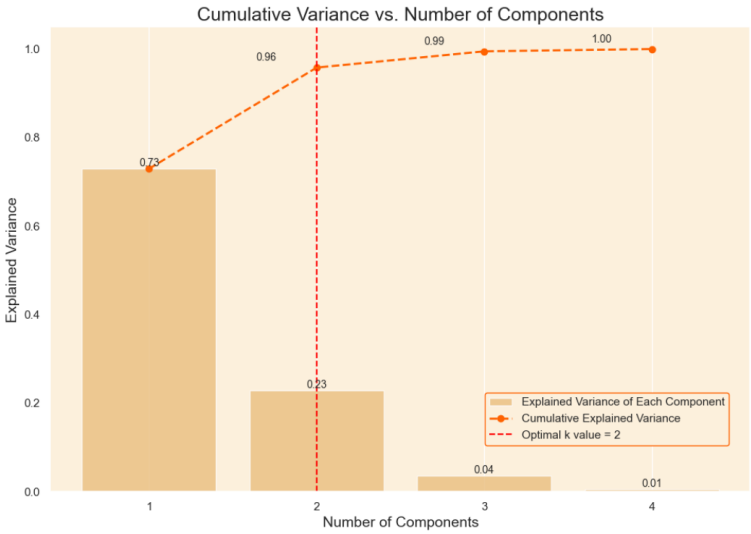
مطابق شکل ۶ و شکل ۷ دیده می‌شود که کلاس Iris-setosa مطابق پیش‌بینی‌ها از توزیع ویژگی‌ها را می‌توان به راحتی از بقیه کلاس‌ها



شکل ۷: بصری‌سازی به کمک PCA

جدا کرد.

برای بررسی اینکه می‌توان از روش‌های کاهش ابعاد استفاده کرد یا نه از متد explained-variance-ratio-. استفاده می‌کنیم تا ببینیم تا کدام یک از Principal Components ها برای توصیف داده‌ها مناسب‌تر است (از نظر تعداد). سپس با چک کردن مقادیر مختلف Principal Components که می‌تواند از ۱ تا ۳ باشد. این نمودار در شکل ۸ آمده است. همان‌طور که از این نمودار مشخص است با وجود دو ویژگی می‌توانیم ۹۶ درصد از واریانس کل را توصیف کنیم. و با ۳ ویژگی می‌توانیم ۹۹ درصد از واریانس را توصیف کنیم.



شکل ۸: واریانس توضیح داده شده با توجه به Principal Components

علاوه بر استناد به نمودار شکل ۸ می‌توان از ماتریس همبستگی در شکل ۵ برداشت کرد که می‌توان یکی از ویژگی‌ها را به علت همبستگی زیاد با یکی دیگر از ویژگی‌ها حذف کرد.

۲.۱

داده‌ها را به دو بخش تقسیم می‌کنیم و ۲۰ درصد از داده‌ها را به تست اختصاص می‌دهیم. سپس داده‌ها را به کمک Standard Scaler، scale می‌کنیم. سپس مطابق کد زیر مدل را با کرنل خطی می‌سازیم.

```
X = from sklearn.svm import SVC
svm_linear = SVC(kernel='linear', random_state=64)
svm_linear.fit(X_train, y_train)

y_pred = svm_linear.predict(X_test)
```

در شکل ۹ نتایج طبقه‌بندی دیده می‌شود. مشاهده می‌شود که دقت کلی ۹۷ درصد است. و تنها در طبقه‌بندی یک کلاس دچار خطا شده است.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	10
1	1.00	0.90	0.95	10
2	0.91	1.00	0.95	10
accuracy			0.97	30
macro avg	0.97	0.97	0.97	30
weighted avg	0.97	0.97	0.97	30

شکل ۹: نتایج طبقه‌بند SVM با کرنل خطی

ماتریس در هم ریختگی نیز در شکل ۱۰ آورده شده است. همانطور که دیده می‌شود طبقه‌بند یک نمونه از کلاس ۱ را در کلاس ۲ طبقه‌بندی کرده و در سایر نمونه‌های تست به خوبی عمل می‌کند.

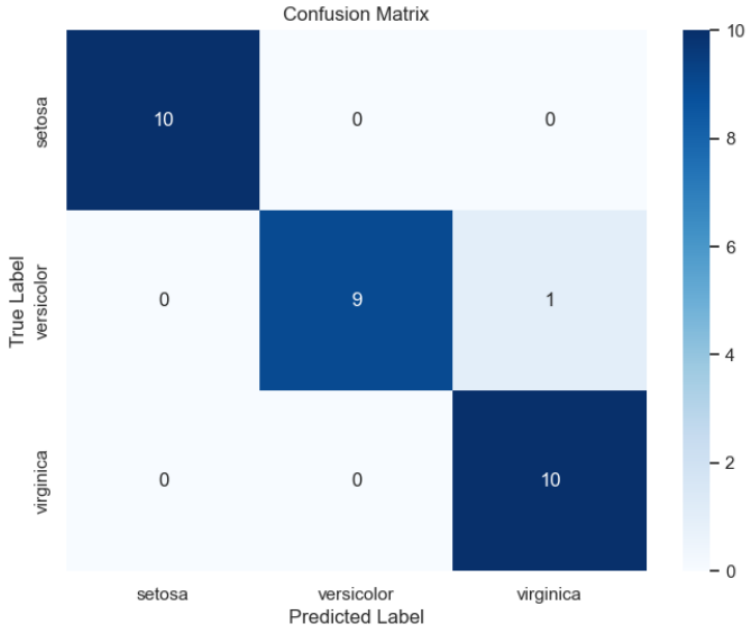
برای رسم مرزهای تصمیم‌گیری می‌توانیم از PCA استفاده کنیم. این ناحیه در شکل ۱۱ آورده شده است.

همانطور که در شکل ۱۱ دیده می‌شود، ناحیه تصمیم‌گیری به صورت خطی جدا شده است.

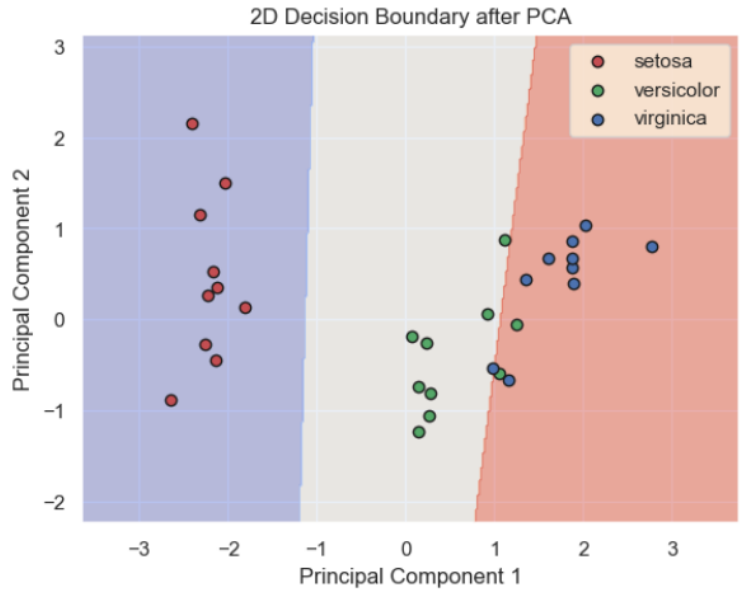
۳.۱

به صورت زیر عمل می‌کنیم و دقت با توجه به هر درجه و همینطور ماتریس درهم ریختگی را برای هر درجه نشان می‌دهیم.

```
iris = datasets.load_iris()
X = iris.data
y = iris.target

scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=64)

```



شکل ۱۰: ماتریس درهم ریختگی طبقه‌بند SVM با کرنل خطی



شکل ۱۱: ناحیه تصمیم‌گیری طبقه‌بند SVM با کرنل خطی با کاهش بعد

```
classifiers = [SVC(kernel='poly', degree=d, C=0.5) for d in range(1, 11)]

accuracies = []
degrees = range(1, 11)
conf_matrices = []

```

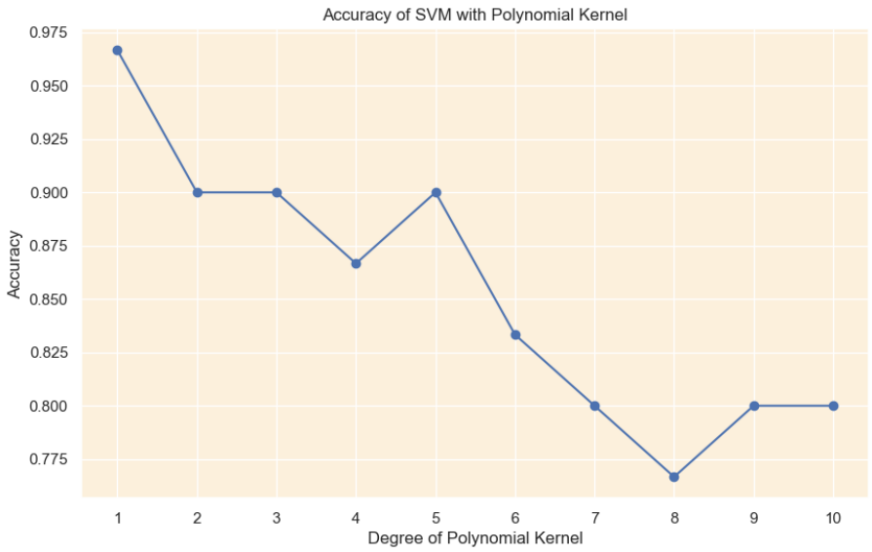
```
for clf in classifiers:
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)
    conf_matrices.append(confusion_matrix(y_test, y_pred))

for degree, accuracy in zip(degrees, accuracies):
    print(f'SVC with polynomial (degree {degree}) kernel: Accuracy = {accuracy:.2f}')

plt.figure(figsize=(10, 6))
plt.plot(degrees, accuracies, marker='o', linestyle='-', color='b')
plt.title('Accuracy of SVM with Polynomial Kernel')
plt.xlabel('Degree of Polynomial Kernel')
plt.ylabel('Accuracy')
plt.xticks(degrees)
plt.grid(True)
plt.show()

fig, axes = plt.subplots(5, 2, figsize=(15, 20))
fig.subplots_adjust(hspace=0.5, wspace=0.3)
for i, (conf_matrix, degree, ax) in enumerate(zip(conf_matrices, degrees, axes.flatten())):
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False, ax=ax)
    ax.set_title(f'Poly Degree {degree}\nAccuracy: {accuracies[i]:.2f}')
    ax.set_xlabel('Predicted')
    ax.set_ylabel('True')
plt.show()
```

علاوه بر این در شکل ۱۲ نمودار مربوط به دقت با توجه به درجه آمده است.

در شکل ۱۳ نیز ماتریس درهم ریختگی به ازای تمامی درجات از ۱ تا ۱۰ آمده است.

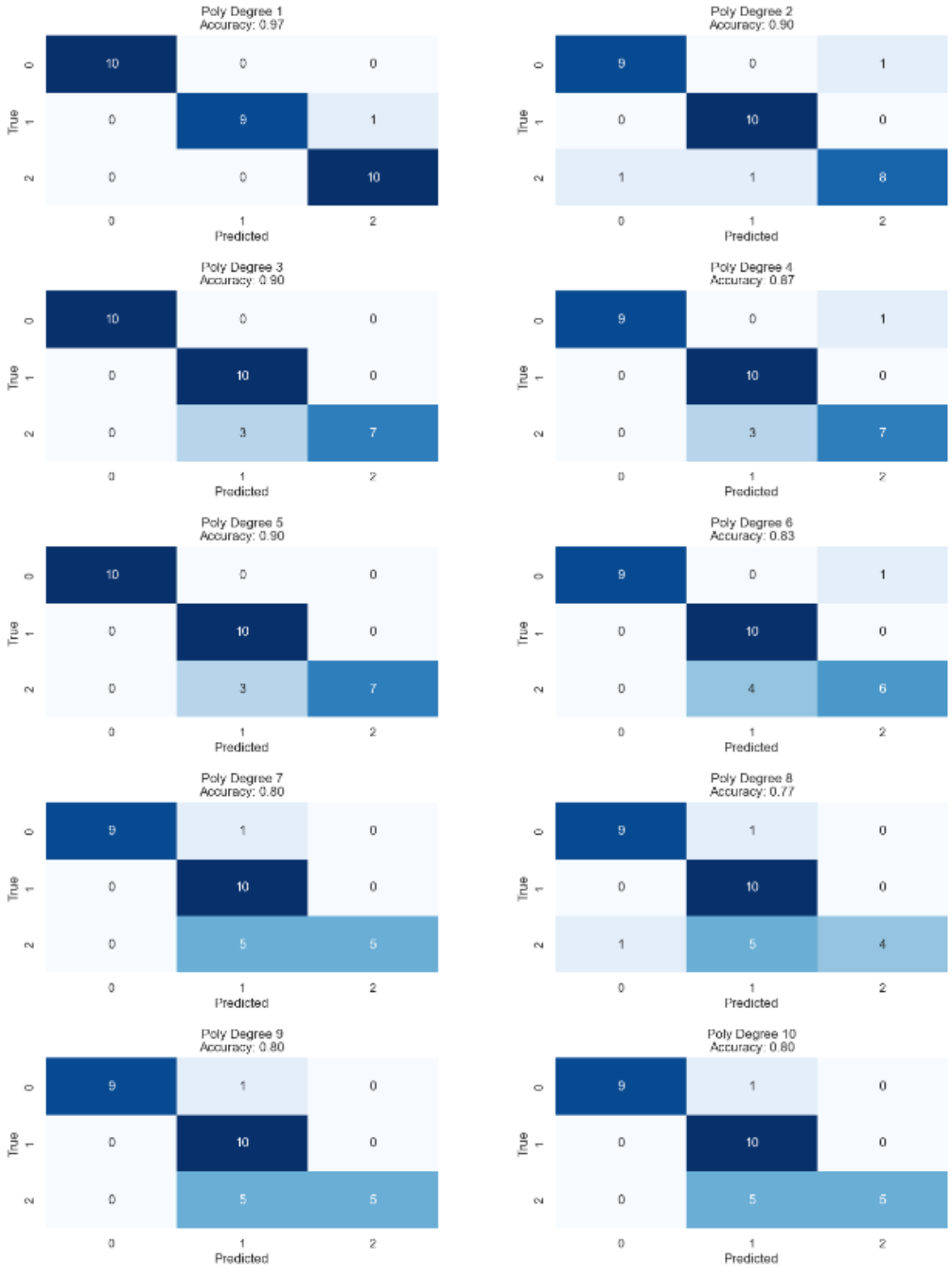
با توجه به شکل ۱۲ دیده می‌شود که با زیاد شدن درجات poly دقت طبقه‌بندی کم و کمتر می‌شود به طوری که در حالت درجه ۱ و به صورت خطی بهترین عملکرد را از مدل شاهد هستیم و با بالا رفتن درجه می‌بینیم که دقت ما کمتر می‌شود به طوری که در درجه ۱۰ دقت به حدود ۷۷ درصد می‌رسد که اصلا قابل قبول نیست. این موضوع را به کمک بصری‌سازی و کمک از PCA برای به تصویر کشیدن این عملکرد مدل طبقه‌بند و رسم ناحیه‌های تصمیم‌گیری به کمک کد زیر و استفاده از مش و کانتور استفاده می‌کنیم.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

```



شکل ۱۲: نمودار دقت با توجه به درجه کرنل

```python
def make_meshgrid(x, y, h=.02):
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy

def plot_contours(ax, clf, xx, yy, **params):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Standardize the data
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Reduce dimensions with PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Create a DataFrame with PCA components and the target
```

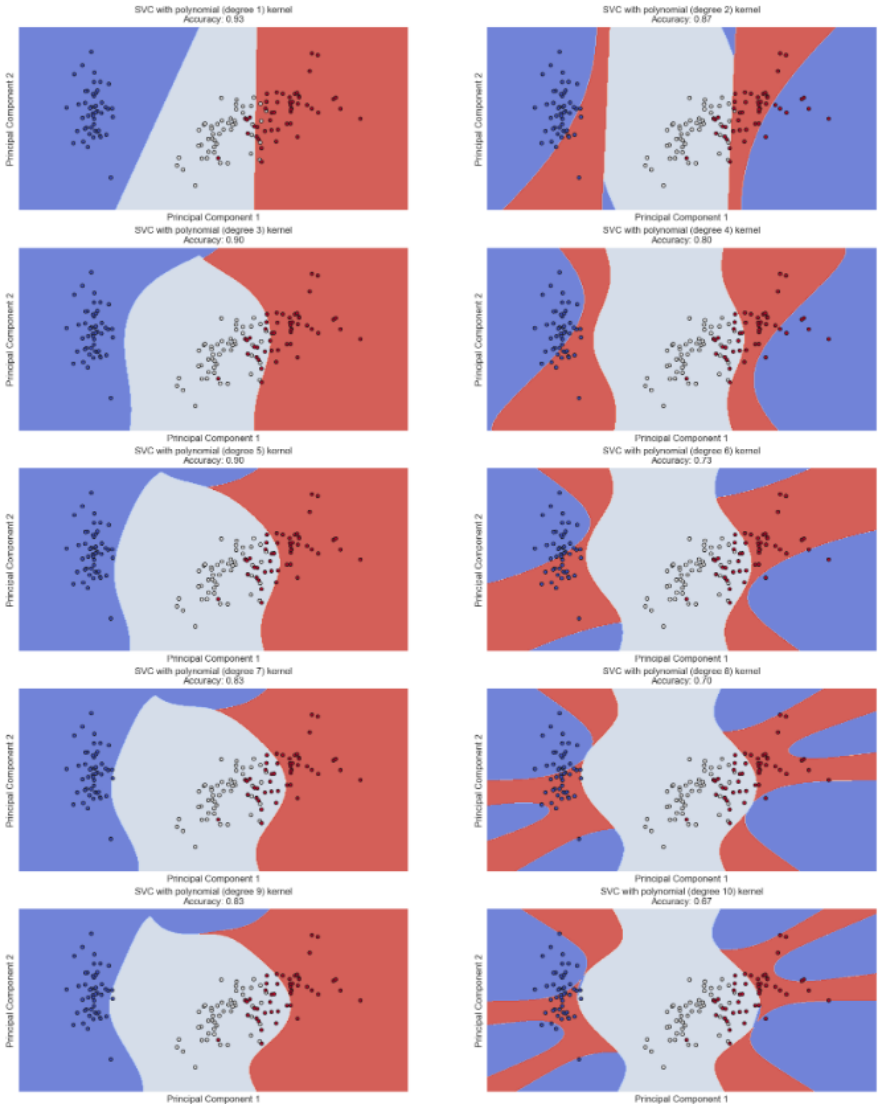


شکل ۱۳: ماتریس درهم ریختگی با توجه به درجه کرنل

```
df_pca = pd.DataFrame(data=X_pca, columns=['PC1', 'PC2'])
df_pca['target'] = y

# Extract X and y from the DataFrame
X = df_pca[['PC1', 'PC2']].values
y = df_pca['target'].values
```

```
44
45 # Split the dataset into training and testing sets
46 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=64)
47
48 # Define classifiers
49 classifiers = [SVC(kernel='poly', degree=d, C=0.5) for d in range(1, 11)]
50
51 # Titles for the plots
52 titles = [f'SVC with polynomial (degree {d}) kernel' for d in range(1, 11)]
53
54 fig, sub = plt.subplots(5, 2, figsize=(20, 25))
55 plt.subplots_adjust(wspace=0.2, hspace=0.2)
56
57 X0, X1 = X[:, 0], X[:, 1]
58 xx, yy = make_meshgrid(X0, X1)
59
60 accuracies = []
61
62 for clf, title, ax in zip(classifiers, titles, sub.flatten()):
63     clf.fit(X_train, y_train)
64     y_pred = clf.predict(X_test)
65     accuracy = accuracy_score(y_test, y_pred)
66     accuracies.append(accuracy)
67
68     plot_contours(ax, clf, xx, yy, cmap=plt.cm.coolwarm, alpha=0.8)
69     ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
70     ax.set_xlim(xx.min(), xx.max())
71     ax.set_ylim(yy.min(), yy.max())
72     ax.set_xlabel('Principal Component 1')
73     ax.set_ylabel('Principal Component 2')
74     ax.set_xticks(())
75     ax.set_yticks(())
76     ax.set_title(f"{title}\nAccuracy: {accuracy:.2f}")
77
78 plt.show()
79
80 # Print accuracies
81 for title, accuracy in zip(titles, accuracies):
82     print(f"{title}: {accuracy:.2f}")
```

نتایج به صورت شکل ۱۴ خواهد بود. همان‌طور که از این شکل پیداست، بهترین عملکرد جدا کردن کلاس‌ها و طبقه‌بندی در degree 1 = رخ می‌دهد و با زیاد کردن درجه صرفا به پیچیدگی مدل اضافه شده و مدل به سمت overfit شدن و یادگرفتن نویزها می‌رود که هیچ، عملکرد طبقه‌بندی آن هم ضعیف‌تر می‌شود. به طوری کلی از شکل ۱۴ مشخص است که با بالا رفتن درجه، انحنای نواحی تصمیم‌گیری بیشتر می‌شود و مدل بهتر می‌تواند پیچیدگی‌های مربوط به داده را یاد بگیرد. اما با توجه به نوع داده ما در اینجا، بهترین نتیجه با طبقه‌بندی خطی که از درجه ۱ بدست می‌آید حاصل می‌شود و در نتیجه بهترین دقت را در این مدل داریم.



شکل ۱۴: ناحیه تصمیم‌گیری طبقه‌بند با توجه به درجات مختلف poly

در مرحله بعد به کمک imageio و کنار هم قرار دادن نمودارهای شکل ۱۴ یک فایل gif از آن‌ها درست می‌کنیم. برای این کار مطابق کد زیر تصویر هر حالت را ذخیره می‌کنیم.

```
import imageio
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from IPython.display import FileLink
from google.colab import drive

drive.mount('/content/drive')

def make_meshgrid(x, y, h=.02):
    x_min, x_max = x.min() - 1, x.max() + 1
```

```
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy

def plot_contours(ax, clf, xx, yy, **params):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out


image_files = []
for degree in range(1, 11):
    clf = SVC(kernel='poly', degree=degree, C=0.5)
    clf.fit(X_train, y_train)

    fig, ax = plt.subplots(figsize=(10, 8))
    plot_contours(ax, clf, xx, yy, cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('Principal Component 1')
    ax.set_ylabel('Principal Component 2')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(f'SVC with polynomial (degree {degree}) kernel')

    filename = f"svc_poly_degree_{degree}.png"
    plt.savefig(filename)
    image_files.append(filename)
    plt.close()

images = []
for filename in image_files:
    images.append(imageio.imread(filename))

# Save GIF to Google Drive
gif_path = '/content/drive/MyDrive/ML2024/MP3/Q3/poly.gif'
imageio.mimsave(gif_path, images, duration=2)  # Duration in seconds for each frame
```

سپس این تصاویر را با فرمت gif به هم می‌چسبانیم و مدت هر فریم را ۲ ثانیه تعریف می‌کنیم.

در نهایت از این لینک گیف (gif) می‌توانید نتیجه نهایی را مشاهده کنید.

۴.۱

به طور کلی کد سه بخش دارد که شامل الگوریتم SVM، بخش بعدی شامل تعمیم این الگوریتم به یک الگوریتم چندکلاسه به روش One vs Rest و بخش آخر شامل بصری‌سازی الگوریتم است.

در قسمت بعد به سراغ ورودی‌های کلاس SVM می‌رویم. دیده می‌شود که باید داده‌ها و تارگت و نوع و پارامترهای کرنل را به این کلاس بدهیم. در نهایت این کلاس به ما مواردی از قبیل y prediction و همینطور ثابت‌ها و نوع کرنل را برمی‌گرداند. علاوه بر این در این کلاس از کتابخانه CVX و از این بهینه‌ساز استفاده شده است.

در مرحله خاصیت multi class classification به این الگوریتم اضافه می‌شود که همانطور که گفته شد از الگوریتم One vs Rest استفاده می‌کند. این بخش نسبت به کدهای خام تغییری نداشته است.

بخش بعدی بحث بصری‌سازی این الگوریتم است که به کمک کلاس visualize-multiclass-classification1 انجام می‌شود.

در نهایت به کمک این سه کلاس ناحیه‌های تصمیم و دقت را با یک حلقه for رسم می‌کنیم.

```
import cvxopt
def linear_kernel( x1, x2):
    return np.dot(x1, x2)

def polynomial_kernel( x, y, C=1.0, d=3):
    return (np.dot(x, y) + C) ** d

def gaussian_kernel( x, y, gamma=0.5):
    return np.exp(-gamma*np.linalg.norm(x - y) ** 2)

def sigmoid_kernel( x, y, alpha=1, C=0.01):
    a= alpha * np.dot(x, y) + C
    return np.tanh(a)

def SVM1(X, X_t, y, C, kernel_type, poly_params=(1, 4), RBF_params=0.5, sigmoid_params=(1, 0.01))
    :
    kernel_and_params=(kernel_type,poly_params, RBF_params, sigmoid_params,C)
    n_samples, n_features = X.shape
    # Compute the Gram matrix
    K = np.zeros((n_samples, n_samples))
    if kernel_type == 'linear':
        for i in range(n_samples):
            for j in range(n_samples):
                K[i, j] = linear_kernel(X[i], X[j])
    elif kernel_type == 'polynomial':
        for i in range(n_samples):
            for j in range(n_samples):
                K[i, j] = polynomial_kernel(X[i], X[j], poly_params[0], poly_params[1])
    elif kernel_type == 'RBF':
        for i in range(n_samples):
            for j in range(n_samples):
                K[i, j] = gaussian_kernel(X[i], X[j], RBF_params)
```

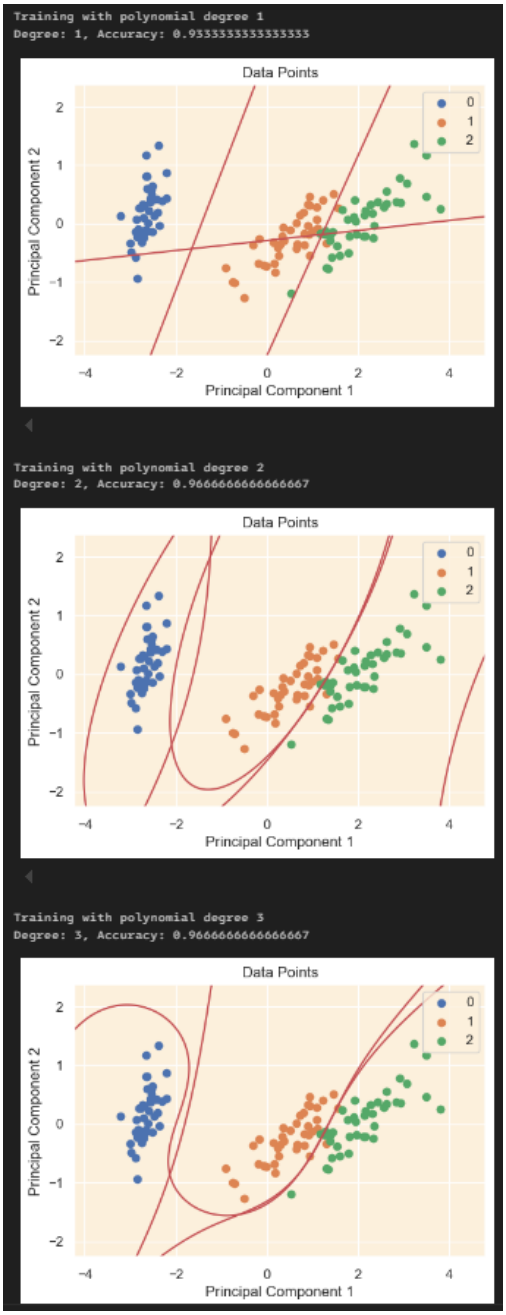
```
    elif kernel_type == 'sigmoid':
        for i in range(n_samples):
            for j in range(n_samples):
                K[i, j] = sigmoid_kernel(X[i], X[j], sigmoid_params[0], sigmoid_params[1])
    else:
        raise ValueError("Invalid kernel type")

    # construct P, q, A, b, G, h matrices for CVXOPT
    P = cvxopt.matrix(np.outer(y, y) * K)
    q = cvxopt.matrix(np.ones(n_samples) * -1)
    A = cvxopt.matrix(y, (1, n_samples))
    b = cvxopt.matrix(0.0)
    G = cvxopt.matrix(np.vstack((np.diag(np.ones(n_samples) * -1), np.identity(n_samples))))
    h = cvxopt.matrix(np.hstack((np.zeros(n_samples), np.ones(n_samples) * C)))
    # solve QP problem
    cvxopt.solvers.options['show_progress'] = False
    solution = cvxopt.solvers.qp(P, q, G, h, A, b)
    # Lagrange multipliers
    a = np.ravel(solution['x'])
    # Support vectors have non zero lagrange multipliers
    sv = a > 1e-5  # some small threshold

    ind = np.arange(len(a))[sv]
    a = a[sv]
    sv_x = X[sv]
    sv_y = y[sv]
    numbers_of_sv=len(sv_y)
    # Bias (For linear it is the intercept):
    bias = 0
    for n in range(len(a)):
        # For all support vectors:
        bias += sv_y[n]
        bias -= np.sum(a * sv_y * K[ind[n], sv])
    bias = bias / (len(a)+0.0001)

    if kernel_type == 'linear':
        w = np.zeros(n_features)
        for n in range(len(a)):
            w += a[n] * sv_y[n] * sv_x[n]
    else:
        w = None

    y_pred=0
    if w is not None:
        y_pred = np.sign(np.dot(X_t, w) + bias)
```

```
    else:
        y_predict = np.zeros(len(X_t))
        for i in range(len(X_t)):
            s = 0
            for a1, sv_y1, sv1 in zip(a ,sv_y, sv_x):
                # a : Lagrange multipliers, sv : support vectors.
                # Hypothesis: sign(sum^S a * y * kernel + b)

                if kernel_type == 'linear':
                    s += a1 * sv_y1 * linear_kernel(X_t[i], sv1)
                if kernel_type=='RBF':
                    s += a1 * sv_y1 * gaussian_kernel(X_t[i], sv1, RBF_params)   # Kernel trick.
                if kernel_type == 'polynomial':
                    s += a1 * sv_y1 * polynomial_kernel(X_t[i], sv1, poly_params[0], poly_params
    [1])
                if kernel_type == 'sigmoid':
                    s=+ a1 * sv_y1 *sigmoid_kernel( X_t[i],  sv1, sigmoid_params[0],
    sigmoid_params[1])
            y_predict[i] = s
        y_pred = np.sign(y_predict + bias)

    return w, bias, solution,a, sv_x, sv_y, y_pred, kernel_and_params



def multiclass_svm(X, X_t, y, C, kernel_type, poly_params=(1, 4), RBF_params=0.5, sigmoid_params
    =(1, 0.01)):
    class_labels = list(set(y))

    classifiers = {}
    w_catch = {}  # catching w, b only for plot part
    b_catch = {}
    a_catch = {}
    sv_x_catch = {}
    sv_y_catch = {}

    for i, class_label in enumerate(class_labels):
        binary_y = np.where(y == class_label, 1.0, -1.0)
        w, bias, solution, a, sv_x, sv_y, prediction, kernel_and_params = SVM1(X, X_t, binary_y,
    C, kernel_type, poly_params, RBF_params, sigmoid_params)
        classifiers[class_label] = (w, bias, a, sv_x, sv_y, kernel_and_params)
        w_catch[class_label] = w
        b_catch[class_label] = bias
        a_catch[class_label] = a
        sv_x_catch[class_label] = sv_x
```

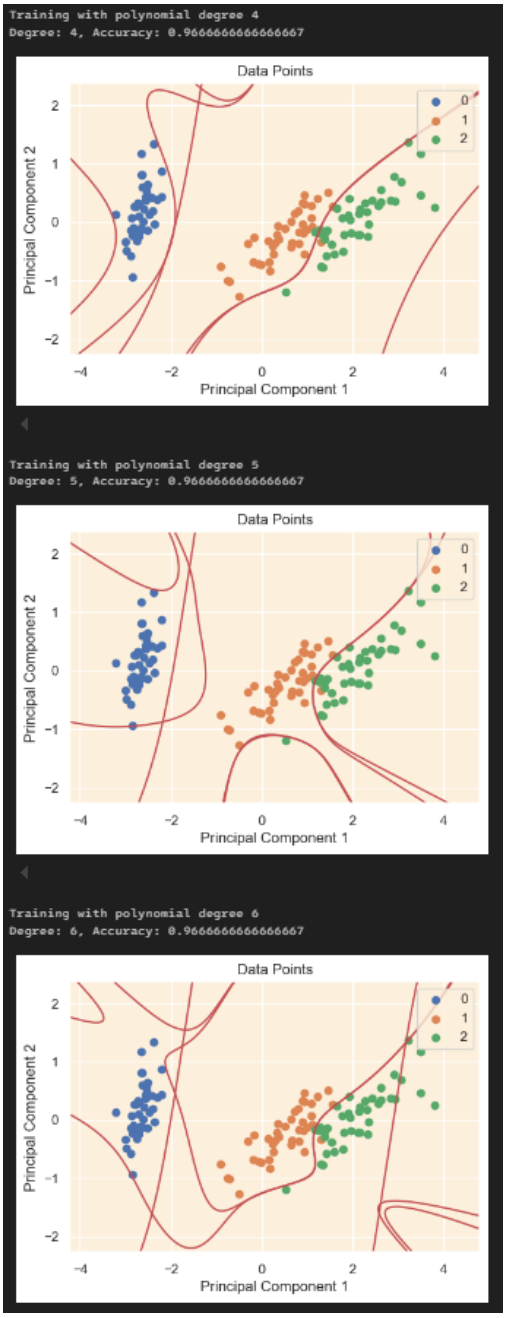
```
        sv_y_catch[class_label] = sv_y

    def decision_function(X_t):
        decision_scores = np.zeros((X_t.shape[0], len(class_labels)))
        for i, label in enumerate(class_labels):
            w, bias, a, sv_x, sv_y, kernel_and_params = classifiers[label]
            if w is not None:
                decision_scores[:, i] = np.dot(X_t, w) + bias
            else:
                decision_values = np.zeros(X_t.shape[0])
                for j in range(X_t.shape[0]):
                    s = 0
                    for a1, sv_y1, sv1 in zip(a, sv_y, sv_x):
                        if kernel_type == 'linear':
                            s += a1 * sv_y1 * linear_kernel(X_t[j], sv1)
                        elif kernel_type == 'RBF':
                            s += a1 * sv_y1 * gaussian_kernel(X_t[j], sv1, RBF_params)
                        elif kernel_type == 'polynomial':
                            s += a1 * sv_y1 * polynomial_kernel(X_t[j], sv1, poly_params[0],
    poly_params[1])
                        elif kernel_type == 'sigmoid':
                            s += a1 * sv_y1 * sigmoid_kernel(X_t[j], sv1, sigmoid_params[0],
    sigmoid_params[1])
                    decision_values[j] = s
                decision_scores[:, i] = decision_values + bias
        return np.argmax(decision_scores, axis=1), kernel_and_params, w_catch, b_catch,
    classifiers

    return decision_function(X_t)

def visualize_multiclass_classification1(X_train, y_train1, kernel_type, trainset, classifiers,
    class_labels, w_stack, b_stack, epsilon=1e-10):
    plt.figure(figsize=(6, 4))
    for i, target_name in enumerate(class_labels):
        plt.scatter(X_train[y_train1 == i, 0], X_train[y_train1 == i, 1], label=target_name)

    if kernel_type == 'linear':
        for i in range(len(class_labels)):
            w = w_stack[i]
            bias = b_stack[i]
            x_points = np.linspace(np.min(X_train[:, 0]) - 1, np.max(X_train[:, 0]) + 1, 200)
            y_points = -(w[0] / (w[1] + epsilon)) * x_points - bias / (w[1] + epsilon)
            plt.plot(x_points, y_points, c='r', label='Decision Boundary')

    elif kernel_type == 'polynomial':
```

```
        x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
        y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
        xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200), np.linspace(y_min, y_max, 200))
        Z = np.zeros(xx.shape)
        for i in range(len(class_labels)):
            Z = np.zeros(xx.shape)
            for j in range(xx.shape[0]):
                for k in range(xx.shape[1]):
                    sample_point = np.array([xx[j, k], yy[j, k]])
                    decision_value = 0
                    w, bias, a, sv_x, sv_y, kernel_and_params = classifiers[i]
                    for a1, sv_y1, sv1 in zip(a, sv_y, sv_x):
                        decision_value += a1 * sv_y1 * polynomial_kernel(sample_point, sv1, C=
    kernel_and_params[1][0], d=kernel_and_params[1][1])
                    decision_value += bias
                    Z[j, k] = decision_value
            plt.contour(xx, yy, Z, levels=[0], colors='r')

    if trainset:
        plt.title('Data Points')
    else:
        plt.title('Data Points on Test Set')

    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.legend()
    plt.xlim(np.min(X_train[:, 0]) - 1, np.max(X_train[:, 0]) + 1)
    plt.ylim(np.min(X_train[:, 1]) - 1, np.max(X_train[:, 1]) + 1)
    plt.show()

iris = datasets.load_iris()
X = iris.data
y = iris.target

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=64)

accuracies = []

for degree in range(1, 11):
    print(f"Training with polynomial degree {degree}")
    predictions, kernel_and_params, w_catch, b_catch, classifiers = multiclass_svm(
        X_train, X_test, y_train, C=1.0, kernel_type='polynomial', poly_params=(1.0, degree)
```

```
    )
    accuracy = accuracy_score(y_test, predictions)
    accuracies.append(accuracy)
    print(f"Degree: {degree}, Accuracy: {accuracy}")

    visualize_multiclass_classification1(X_train, y_train, 'polynomial', True, classifiers, np.
    unique(y_train), w_catch, b_catch)
```



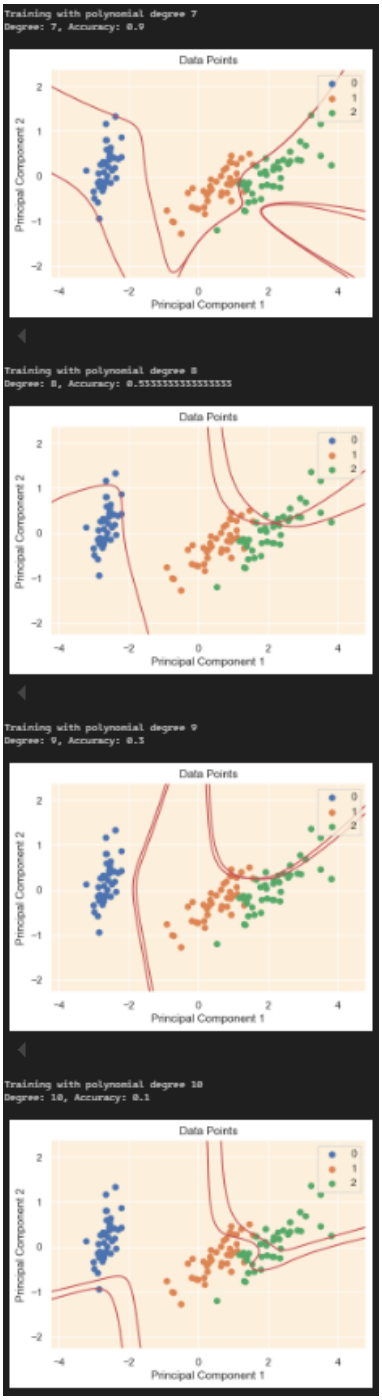
شکل ۱۵: polynomial ۳ درجه اول به صورت دستی



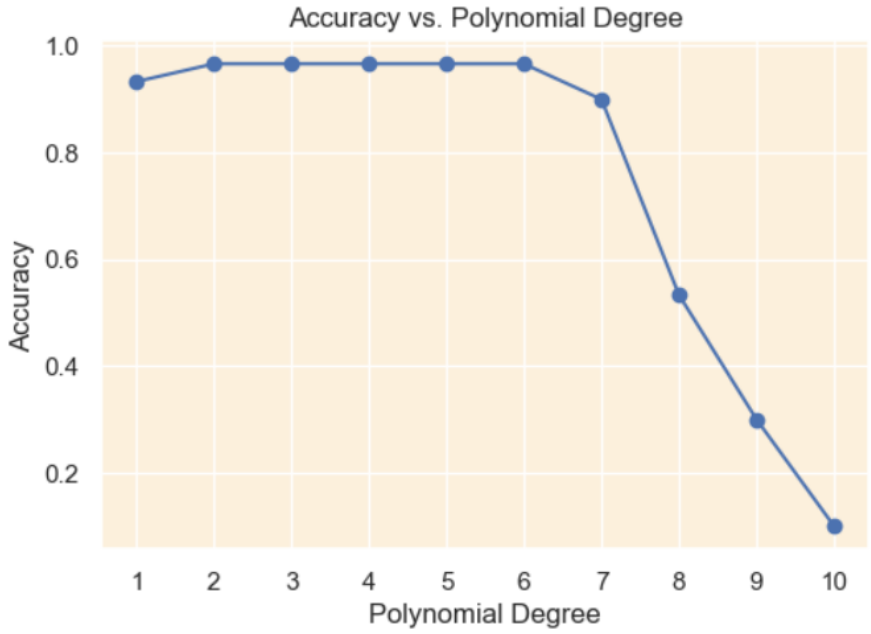
شکل ۱۶: polynomial ۳ درجه دوم به صورت دستی

در شکل ۱۸ نیز میزان دقت با توجه به هر درجه آورده شده که نتایجی مشابه قسمت قبل دارد. با پیچیده شدن طبقه‌بند و افزایش درجه، دقت افت محسوسی داشته است.

در نهایت مانند بخش قبلی می‌توانید از این لینک گیف مربوطه را دانلود کنید.



شکل ۱۷: polynomial ۴ درجه سوم به صورت دستی



شکل ۱۸: مودار دقت با توجه به درجه به صورت دستی

۲ سوال سوم

۱.۲

داده‌های نامتوازن

یکی از چالش‌های اصلی، نامتوازن بودن بین کلاس‌های اکثریت (تراکنش‌های عادی) و اقلیت (تراکنش‌های تقلبی) است. مقاله تأکید می‌کند که روش‌های طبقه‌بندی سنتی با مجموعه داده‌های نامتوازن که در تشخیص تقلب رایج هستند، به سختی کار می‌کنند.

نویز در داده‌ها

بیش‌نمونه‌گیری یک تکنیک برای متوازن کردن نمونه‌های کلاس‌ها است، اما می‌تواند نویز را وارد کند. مقاله یک شبکه عصبی خودرمزگذار نویزگیر (DAE) را پیشنهاد می‌دهد که نه تنها نمونه‌های کلاس اقلیت را بیش‌نمونه‌گیری می‌کند، بلکه مجموعه داده‌ها را نویزگیری و طبقه‌بندی می‌کند.

Denoising AutoEncoder

الگوریتم DAE برای بهبود دقت طبقه‌بندی طراحی شده است، به طوری که با یادگیری حذف نویز و بازسازی ورودی بدون اختلال، به مدل کمک می‌کند تا بهتر تعمیم یابد و در برابر داده‌های خراب مقاوم‌تر باشد.

روش‌های استفاده شده برای حل چالش‌ها

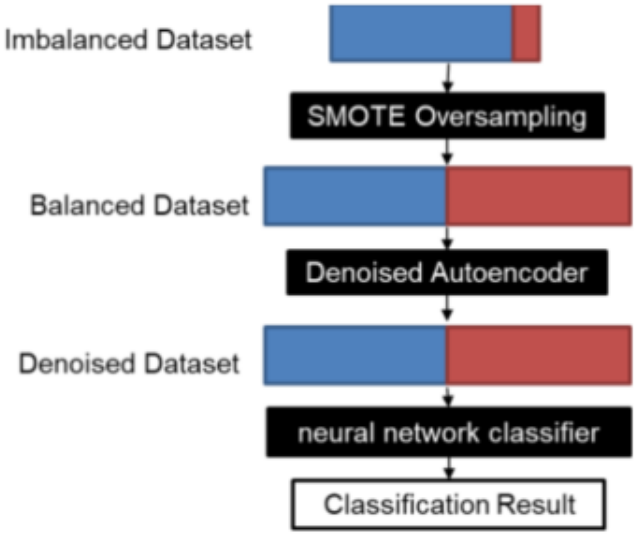
- بیش‌نمونه‌گیری با SMOTE: مجموعه داده را با تولید نمونه‌های کلاس اقلیت متوازن می‌کند.

- شبکه عصبی خودرمزگذار نویزگیر (DAE): به طور همزمان داده‌ها را نویزگیری و طبقه‌بندی می‌کند و دقت تشخیص کلاس اقلیت را بهبود می‌بخشد.
- تحلیل مؤلفه‌های اصلی (PCA): ابعاد داده را کاهش می‌دهد و ویژگی‌های مربوطه را انتخاب می‌کند.

مدل پیشنهادی که ترکیبی از بیش‌نمونه‌گیری و خودرمزگذارهای نویزگیر است، بهبودهای قابل توجهی در تشخیص تراکنش‌های تقلبی نسبت به روش‌های سنتی نشان می‌دهد. نتایج ارزیابی نشان می‌دهد که مدل نرخ‌های یادآوری بالاتری را به دست می‌آورد، به این معنی که بخش بزرگتری از تراکنش‌های تقلبی را به طور دقیق شناسایی می‌کند در حالی که دقت کلی قابل قبولی را حفظ می‌کند.
این راه‌حل‌ها و روش‌ها به کاهش چالش‌های ذاتی در توسعه مدل‌های تشخیص تقلب قدرتمند و دقیق کمک می‌کنند، به خصوص در زمینه مجموعه داده‌های نامتوازن و نویزی.

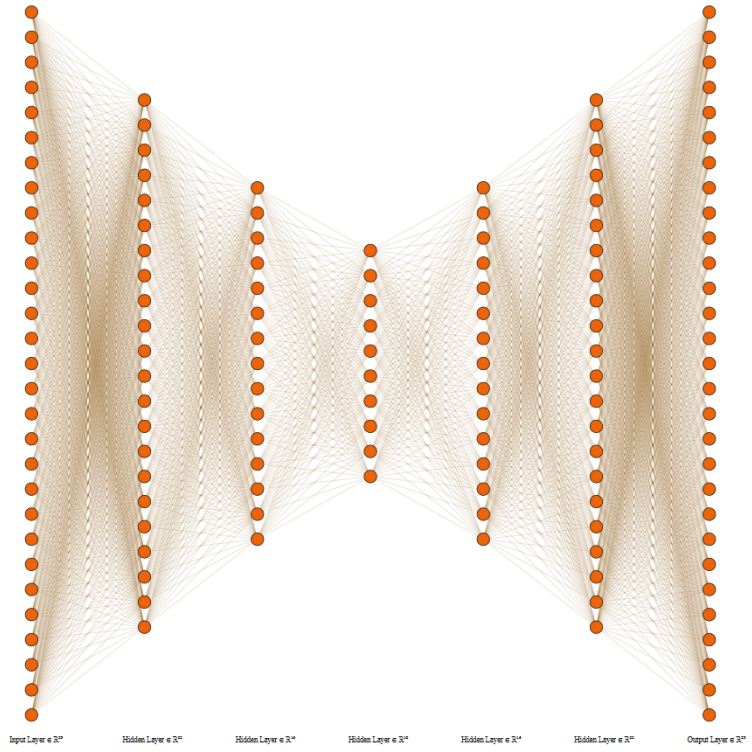
۲.۲

با توجه به توضیحات مقاله فلوچارت کلی به صورت زیر است.



شکل ۱۹: فلوچارت مدل طبقه‌بندی

مشاهده می‌شود که ابتدا با استفاده از Oversampling داده‌ها بالانس می‌شوند و در مرحله بعد وارد شبکه Autoencoder می‌شود که معماری آن به صورت زیر است.

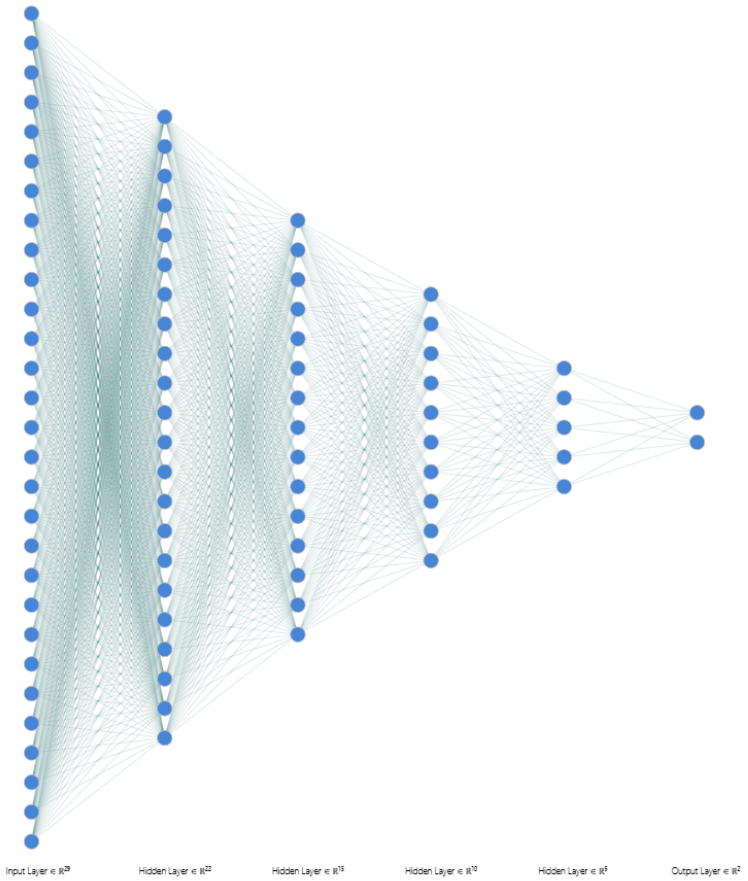


شکل ۲۰: معماری شبکه Autoencoder

به طور کلی Autoencoder ها ساختاری به صورت بالا دارند که تعداد ورودی و خروجی‌ها برابر و در لایه میانی به صورت bottle neck هستند. این ساختار در کاهش ابعاد داده و فشردهسازی کاربرد زیادی دارد.

در این مورد از DAE استفاده شده که برای حذف نویز و مواجهه با داده خراب مدل را آموزش می‌دهد و موجب robust شدن مدل می‌شود.

در مرحله بعد خروجی Autoencoder به یک شبکه طبقه‌بندی با ساختار زیر وارد می‌شود.



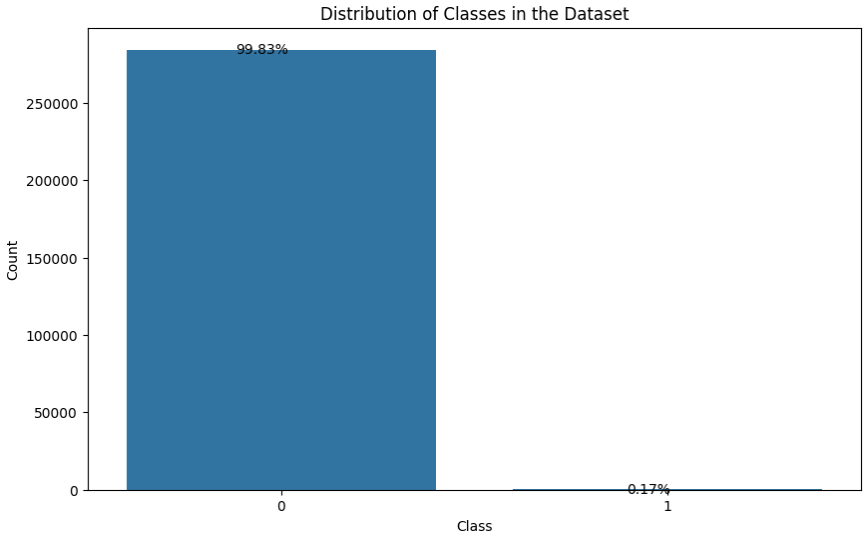
شکل ۲۱: معماری شبکه عصبی برای طبقه‌بندی

در نهایت خروجی این شبکه به یک SoftMax Cross Entropy Loss Function وارد می‌شود تا کار طبقه‌بندی به کمک آن صورت گیرد.

مقاله در مورد دلیل انتخاب تعداد لایه‌ها و نودها صحبتی نکرده‌است.

۳.۲

حال به پیاده‌سازی این مدل می‌پردازیم. ابتدا پراکندگی برچسب‌ها را مورد بررسی قرار می‌دهیم. این پراکندگی به صورت زیر است:



شکل ۲۲: نمودار پراکندگی برچسب‌ها

همانطور که مشاهده می‌شود برچسب داده به شدت نامتوازن است. حال به پیش‌پردازش داده می‌پردازیم. در ابتدا با توجه به گفته مقاله، ستون "Time" را حذف می‌کنیم و ستون "Amount" را نرمال می‌کنیم. با توجه به گفته مقاله سایر ستون‌ها نیاز به پیش‌پردازش ندارند و از فرایند PCA استخراج شده‌اند.

سپس به تقسیم داده می‌پردازیم. در این قسمت برای حفظ نسبت تقسیم کلاس‌ها از stratify در دستور train test split استفاده می‌کنیم و داده را به نسبت ۶۰، ۲۰ و ۲۰ تقسیم می‌کنیم.

```
Training set class distribution:
 [170589.    295.]

Validation set class distribution:
 [56863.    98.]

Testing set class distribution:
 [56863.    99.]
```

شکل ۲۳: پراکندگی برچسب‌ها

در مرحله بعد با استفاده از روش SMOTE که یک روش Oversampling برای متوازن کردن داده است، داده را متوازن می‌کنیم. در ابتدا این کار را برا آستانه 0.5 انجام می‌دهیم که موجب برابر شدن تعداد نمونه‌های هر دو کلاس می‌شود. باید توجه کرد که این پروسه صرفا

باستی روی دسته آموزش صورت گیرد.

حال Denoising Autoencoder را پیاده‌سازی می‌کنیم.

```
# Adding Gaussian noise to the data
def add_noise(data, noise_factor=0.2):
    noisy_data = data + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=data.shape)
    noisy_data = np.clip(noisy_data, 0., 1.)
    return noisy_data

X_train_noisy = add_noise(X_train_res)
X_valid_noisy = add_noise(X_valid)

# Define the autoencoder model
input_dim = X_train_res.shape[1]
encoding_dim = 10

input_layer = Input(shape=(input_dim,))
encoder = Dense(encoding_dim, activation="relu")(input_layer)
encoder = Dense(22, activation="relu")(encoder)
encoder = Dense(15, activation="relu")(encoder)
encoder = Dense(encoding_dim, activation="relu")(encoder)
encoder = Dense(15, activation="relu")(encoder)
encoder = Dense(22, activation="relu")(encoder)
decoder = Dense(input_dim, activation='sigmoid')(encoder)

autoencoder = Model(inputs=input_layer, outputs=decoder)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')

checkpoint = ModelCheckpoint('best_autoencoder.h5', monitor='val_loss', save_best_only=True, mode
    ='min', verbose=1)

autoencoder.fit(X_train_noisy, X_train_res,
                epochs=20,
                batch_size=256,
                shuffle=True,
                validation_data=(X_valid_noisy, X_valid),
                callbacks=[checkpoint],
                verbose=1)
```

همانطور که در کد مشاهده می‌شود، ابتدا یک نویز گاوسی با پهنای ۱ و حول ۰ با دامنه 0.2 ساخته می‌شود. سپس این نویز را وارد داده آموزش و اعتبارسنجی می‌کنیم.

در مرحله بعد Autoencoder را با توجه به ساختار مقاله پیاده‌سازی می‌کنیم. برای توابع فعالساز از ReLU استفاده می‌کنیم و تنها در لایه آخر sigmoid قرار می‌دهیم. تابع خطا و بهینه‌ساز نیز به ترتیب MSEو ADAM هستند. سایر پارامترها نیز در کد مشخص هستند.

در مرحله بعد خروجی این Autoencoder وارد یک شبکه طبقه‌بند می‌شود که ساختار آن نیز با توجه به مقاله پیاده می‌شود که به صورت زیر است.

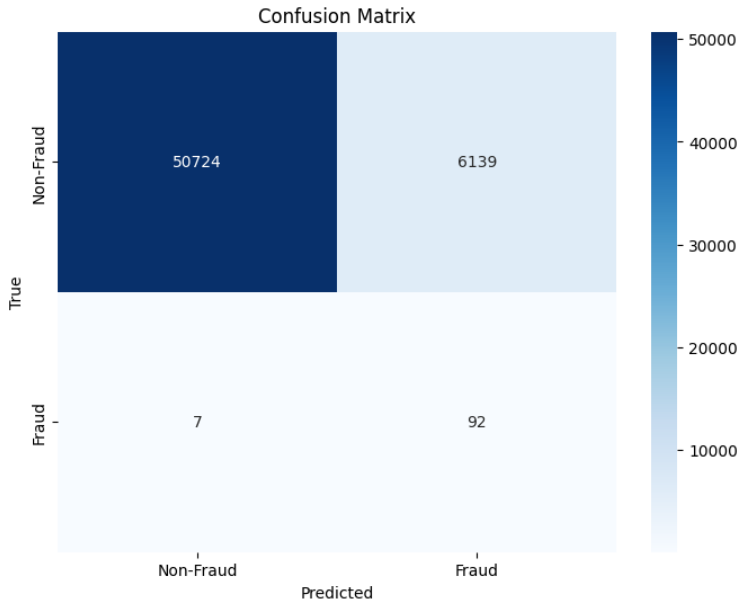
```
autoencoder.load_weights('best_autoencoder.h5')
# Denoise the data
X_train_denoised = autoencoder.predict(X_train_noisy)
X_valid_denoised = autoencoder.predict(X_valid_noisy)

# Define the classifier model
classifier_input = Input(shape=(input_dim,))
classifier_layer = Dense(encoding_dim, activation="relu")(classifier_input)
classifier_layer = Dense(22, activation="relu")(classifier_layer)
classifier_layer = Dense(15, activation="relu")(classifier_layer)
classifier_layer = Dense(10, activation="relu")(classifier_layer)
classifier_layer = Dense(5, activation="relu")(classifier_layer)
classifier_layer = Dense(2, activation='softmax')(classifier_layer)

classifier = Model(inputs=classifier_input, outputs=classifier_layer)
classifier.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
classifier_checkpoint = ModelCheckpoint('best_classifier.h5', monitor='val_loss', save_best_only=
    True, mode='min', verbose=1)

# Train the classifier with the ModelCheckpoint callback
classifier.fit(X_train_denoised, y_train_res,
               epochs=20,
               batch_size=256,
               shuffle=True,
               validation_data=(X_valid_denoised, y_valid),
               callbacks=[classifier_checkpoint],
               verbose=1)
```

نتایج اولیه مدل به صورت زیر است.

```
1781/1781 [==============================] - 3s 1ms/step
Loss: 0.5190405249595642, Accuracy: 0.8921034932136536
1781/1781 [==============================] - 5s 2ms/step
Recall: 0.9292929292929293
```

۴.۲

در این قسمت نتایج را بررسی می‌کنیم.



شکل ۲۴: ماتریس درهمریختگی

	precision	recall	f1-score	support
Non-Fraud	1.00	0.89	0.94	56863
Fraud	0.01	0.93	0.03	99
accuracy			0.89	56962
macro avg	0.51	0.91	0.49	56962
weighted avg	1.00	0.89	0.94	56962

شکل ۲۵: گزارش طبقه‌بندی با معیارهای مختلف

مشاهده می‌شود که که مدل در پیدا کردن برچسب‌های تقلب تقریبا به خوبی عمل کرده‌است و تنها نمونه را تشخیص نداده است ولی تعداد زیادی از نمونه‌های سالم را تقلب تشخیص داده که منجر به پایین بودن درصد در معیار precision شده است. این مشکلی است که معمولا در داده نامتوازن رخ می‌دهد.

در مسائلی که توزیع برچسب‌ها نامتعادل است، استفاده از دقت به تنهایی معمولاً نمایانگر صحیح عملکرد مدل نیست. دلیل آن این است که دقت می‌تواند در مواقعی که یک کلاس به طور قابل توجهی بیشتر از کلاس‌های دیگر است، گمراه‌کننده باشد.

چرا دقت می‌تواند گمراه‌کننده باشد؟

- توزیع نامتعادل: در یک مجموعه داده نامتعادل، کلاس اکثریت بر کلاس اقلیت تسلط دارد. اگر کلاس اکثریت ۹۵% داده‌ها را تشکیل دهد و کلاس اقلیت تنها ۵% باشد، مدلی که همیشه کلاس اکثریت را پیش‌بینی کند، دقت ۹۵% خواهد داشت اما برای شناسایی کلاس اقلیت بی‌فایده خواهد بود.

- مثبت کاذب و منفی کاذب: دقت بین مثبت‌های کاذب و منفی‌های کاذب تمایز قائل نمی‌شود. به عنوان مثال، در تشخیص تقلب، منفی‌های کاذب (تراکنش‌های تقلبی که به عنوان غیرتقلبی پیش‌بینی شده‌اند) اغلب مهم‌تر از مثبت‌های کاذب هستند.

معیارهای جایگزین

برای ارزیابی بهتر عملکرد مدل در مجموعه داده‌های نامتعادل، می‌توان از معیارهای زیر استفاده کرد:

- دقت (Precision): نسبت پیش‌بینی‌های صحیح مثبت به تمام پیش‌بینی‌های مثبت را اندازه‌گیری می‌کند.

$$\text{Precision} = \frac{\text{Positives True}}{\text{Positives True} + \text{Positives False}}$$

دقت بالا نشان می‌دهد که مدل نرخ مثبت کاذب پایینی دارد.

- بازخوانی (Recall): نسبت پیش‌بینی‌های صحیح مثبت به تمام نمونه‌های مثبت واقعی را اندازه‌گیری می‌کند.

$$\text{Recall} = \frac{\text{Positives True}}{\text{Positives True} + \text{Negatives False}}$$

بازخوانی بالا نشان می‌دهد که مدل می‌تواند اکثر نمونه‌های مثبت را شناسایی کند و تعداد منفی‌های کاذب را کاهش دهد.

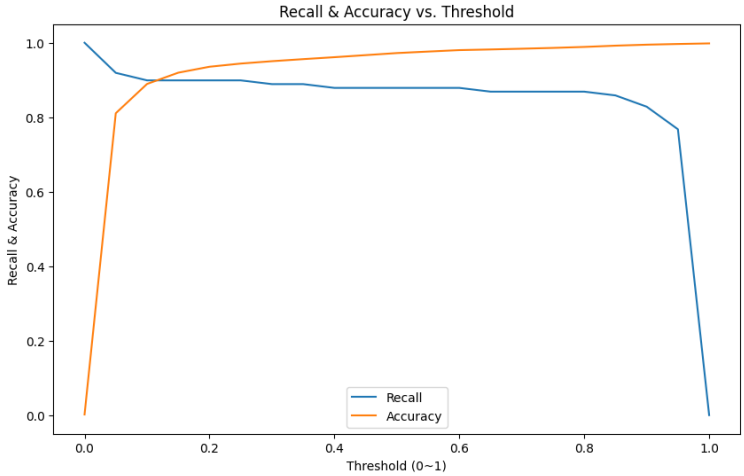
- نمره F۱: میانگین هارمونیک دقت و بازخوانی، که توازنی بین این دو فراهم می‌کند.

$$\text{F۱-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

این معیار زمانی مفید است که نیاز به توازن بین دقت و بازخوانی داریم.

حال مطابق با مقاله با ایجاد یک آستانه تصمیم‌گیری نمودار Recall و Accuracy را رسم می‌کنیم.

```
thresholds = np.arange(0.0, 1.05, 0.05)

recalls = []
accuracies = []
for threshold in thresholds:
    y_pred = (y_pred_prob[:, 1] >= threshold).astype(int)
    recall = recall_score(np.argmax(y_test, axis=1), y_pred)
    accuracy = accuracy_score(np.argmax(y_test, axis=1), y_pred)
    recalls.append(recall)
    accuracies.append(accuracy)

plt.figure(figsize=(10, 6))
plt.plot(thresholds, recalls, label='Recall')
plt.plot(thresholds, accuracies, label='Accuracy')
plt.xlabel('Threshold (0~1)')
plt.ylabel('Recall & Accuracy')
plt.title('Recall & Accuracy vs. Threshold')
plt.legend()
plt.show()
```



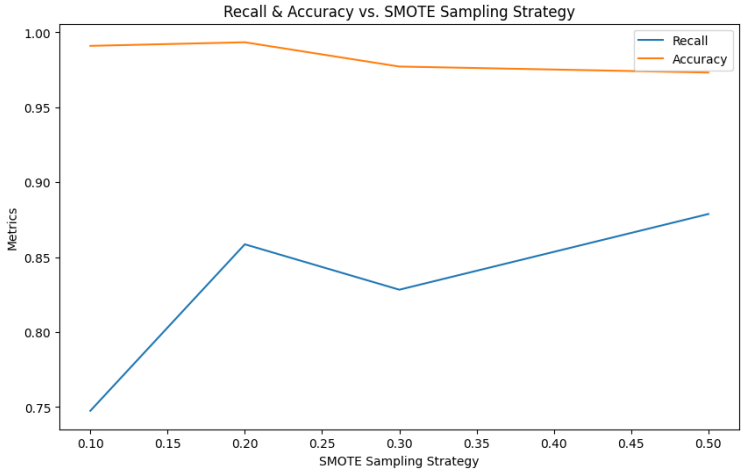
شکل ۲۶: نمودار accuracy در برابر recall

Threshold	Recall Rate	Accuracy
0.2	89.90%	93.56%
0.3	88.89%	95.05%
0.4	87.88%	96.12%
0.5	87.88%	97.23%
0.6	87.88%	98.02%
0.7	86.87%	98.40%
0.8	86.87%	98.88%

همانطور که مشخص است با افزایش مقدار آستانه دقت مدل افزایش می‌یابد و معیار recall کاهش می‌یابد.

۵.۲

حال به پیاده‌سازی مدل با تغییر آستانه Oversampling می‌پردازیم. با انتخاب ۴ آستانه مدل را به همین تعداد آموزش می‌دهیم. نتایج به صورت زیر است.



شکل ۲۷: نمودار accuracy در برابر recall برای آستانه‌های متفاوت oversampling

SMOTE Threshold	Recall Rate	Accuracy
0	0.1	0.747475	0.990976
1	0.2	0.858586	0.993382
2	0.3	0.828283	0.977178
3	0.5	0.878788	0.973193

مشاهده می‌شود این روش باعث بهبود مدل می‌شود و هرچه تعداد نمونه‌های تقلب که گروه اقلیت است را بیشتر می‌کنیم این مدل در پیشبینی آنها بهتر عمل می‌کند هرچند این امر باعث کاهش دقت کلی مدل می‌شود ولی از بایاس شدن مدل روی نمونه غالب جلوگیری می‌کند.

۶.۲

در این قسمت به پیاده‌سازی بدون استفاده از oversampling و denoising autoencoder می‌پردازیم.
نتایج به شرح زیر است.

Classification Report (Unbalanced Data with Noise Added)

	precision	recall	f1-score	support
Non-Fraud	1.00	1.00	1.00	56863
Fraud	0.81	0.76	0.78	99
accuracy			1.00	56962
macro avg	0.90	0.88	0.89	56962
weighted avg	1.00	1.00	1.00	56962

شکل ۲۸: نتایج طبقه‌بندی در حالت دوم

شکل ۲۹: ماتریس درهم‌ریختگی در حالت دوم

با توجه به نتایج مشاهده می‌شود که مدل در این حالت recall پایین‌تری دارد و در تشخیص موارد تقلب ضعیف‌تر عمل می‌کند. اما در حالت کلی در این حالت رفتار کلی متوازن تر است و درصد precision نیز مناسب است هرچند در این مقاله هدف اصلی پیدا کردن نمونه‌های تقلب است و تمرکز اصلی بایستی روی پیدا کردن آنها باشد، در نتیجه ترجیح ما استفاده از denoising autoencoder و smote است تا نتایج recall را بهبود دهیم.

لینک Github

لینک Colab